Redocumentation through design pattern recovery:: an investigation and an implementation

Kim Hyoseob

Redocumentation through Design Pattern Recovery: An Investigation and an Implementation

Hyoseob Kim

PhD Thesis

Department of Computer Science

University of Durham

December 2001

Abstract

In this thesis, two methods are developed in an aid to help users capture valuable design information and knowledge and reuse them. They are the design pattern recovery (DPR) method and pattern-based redocumentation (PBR) method. The DPR method is for matching up metrics of patterns with patterns themselves in order to capture valuable design information. Patterns are used as a container for storing the information. Two new metrics, i.e., *p-value* and *s-value* are introduced. They are obtained by analysing product metrics statistically. Once patterns have been detected from a system, the system can be redocumented using these patterns. Some existing XML (eXtensible Markup Language) technologies are utilised in order to realise the PBR method.

Next, a case study is carried out to validate the soundness and usefulness of the DPR method.

Finally, some conclusions drawn from this research are summarised, and further work is suggested for the researchers in software engineering.

Acknowledgements

A technical work of this size can obviously not be produced without a great deal of help, advice and encouragement from others. A number of people have aided and abetted in its production.

I owe a great debt of gratitude to my supervisor Dr. Cornelia Boldyreff, who was always on hand to listen to my ideas and always willing to contribute her own. Also, she took the time to read the drafts of this thesis and provided invaluable feedback.

I am really grateful to my thesis examiners, Prof. Malcolm Munro and Prof. Ian Sommerville for their constructive suggestions that have helped me improve the content and representation of this thesis.

I really thank Dr. Elizabeth Burd for allowing her valuable time to review the draft of this thesis.

On a personal level, my deepest thanks are due to my parents who have always supported their son emotionally and financially.

Declaration

No part of the material offered has previously been submitted by the author for a degree in the University of Durham or in any other University. All of the work presented here is the sole work of the author and no-one else.

Statement of Copyright

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This chapter gives an introduction to the thesis. First, the typical problem of developing high quality software at low cost is studied. Second, a solution for this problem is proposed. Then, the contributions made during this research are listed, and some criteria for measuring the success of the research are suggested in the next section. Finally, this chapter ends by giving an outline of the thesis.

1.1 The Problem

Originally people had believed that software would not outlive the hardware systems on that it was running. However, as the computing history tells us, many software systems have actually survived beyond their developers' expectation. A typical example

of this is software with the so-called "millennium bug" [121]. When they built their software systems decades ago, they did not think they would need them beyond the 20th century!

Like any other creatures or artificial things, software systems evolve, adapting to changing environments and having their errors fixed as they are detected. Thus, the need for the four kinds of maintenance activities arises to keep our system running. They are *corrective maintenance*, *adaptive maintenance*, *perfective maintenance*, and *preventive maintenance* [105].

Corrective maintenance is concerned with the location and removal of faults in the program. These are errors in what the program actually does according to the current specification; it is not concerned with erroneous output caused due to a change in the specification.

Adaptive maintenance involves the updating of the program due to a change in the environment in which it has to run. This may be a minor change which involves little change in the structure of the program, for example, a change in printed output from English to American spelling, or it may be a major change, such as rewriting the program to run in a distributed fashion on a network.

Perfective maintenance is maintenance resulting from a change in a program's specification. This might be as simple as a change in the format in which a report is required, or as complex as the addition of a different kind of account to a financial

banking program. Perfective maintenance takes up as much as half of a maintenance programmer's time.

Finally, preventive maintenance is the modification to software undertaken to improve some attribute of that software, such as its "quality" or "maintainability" without altering its functional specification.

These four kinds of maintenance are often carried out together so that it may not be easy to distinguish them.

Further, the above maintenance activities are supported by the following four activities [32]:

- restructuring: The transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behaviour, i.e., functionality and semantics.

- redocumentation: The creation or revision of a semantically equivalent representation within the same relative abstraction level. The resulting forms of representation are usually considered alternate views intended for a human audience.

- reengineering: The examination and modification of a system to reconstitute it in a new form and the subsequent implementation of the new form. Sometimes it is difficult to differentiate reengineering from restructuring. The main difference between them is that restructuring keeps the functionality of the subject system

intact, while reengineering changes it usually.

- reverse engineering: The process of analysing an existing system to identify its components and their interrelationships and create representations of the system in another form or at a higher level of abstraction [89]. Reverse engineering is usually undertaken in order to redesign the system for better maintainability or to produce a copy of a system without access to the design from which it was originally produced.

The costs and difficulties involved in software maintenance have been well documented. A major contributor to these costs is the time-consuming process of "program comprehension". Program comprehension is performed during the process of reuse, reengineering, and enhancing existing systems. It is also performed during review or code walk-through of new programs. In a narrow sense, program comprehension means the understanding of program code. In a wide sense, however, it incorporates all aspects of understanding of an application or system. The comprehension process is the sum of a number of understanding techniques [11, 108, 110].

To summarise, program comprehension is one of the first things to be performed before any maintenance activities begin. Because without having a full and correct understanding of a subject system, the above remedial activities can make situations worse rather than actually improving it.

The software engineering discipline is a comparatively new branch of the computer

science, mainly attempting to find efficient ways of building quality software. The term, software engineering itself was first coined at the First NATO Software Engineering Conference held at Garmisch in 1968 [90]. Since then, whether software engineering is an engineering discipline in the true sense has been always arguable. Among the many reasons why software engineering has failed to become a true engineering branch, people agree that software engineering lacks *design reuse* and *measurement* compared to other mature engineering disciplines such as chemical engineering and electronics [66, 98].

First, design reuse is not actively practised in the software engineering discipline. Generally, engineering designs can be classified into two kinds: *routine designs* and *innovative designs* [66]. The former ones involve solving familiar problems whereas the latter ones indicate finding novel solutions to unfamiliar problems [117]. Therefore, it is right to say that innovative designs are the ones that needs greater attention from developers when systems are built, while it is desirable to maximise the benefits from reusing existing solutions, i.e., routine ones. Other names for these two kinds of designs are state-of-the-practice and state-of-the-art designs, respectively. Obviously, innovative designs take more time and cost more than routine ones to develop. Figure 1.1 shows how engineering designs are extensively reused. As in other engineering disciplines, innovative designs are much rarer than routine ones within software engineering. Jones reports that less than 15% of the software developed is innovative [58]. This means that in theory new software can be built out of already existing parts of

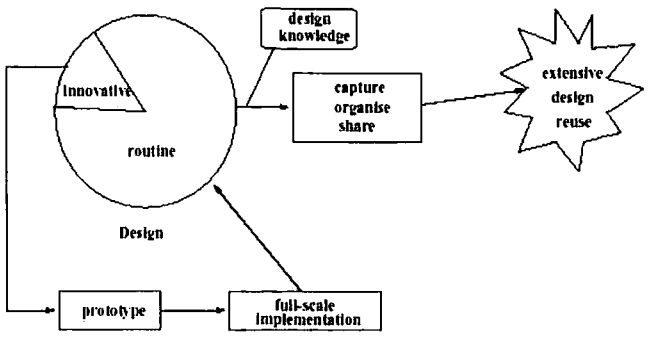

Figure 1.1: Engineering design (Kogut, 1995)

systems and adding only minor new features to them. However, it is also true that because of the difficulties of adapting those parts to the new environment, systems are usefully built from scratch incurring high cost. This certainly gives a plenty of room for furthering reuse practices. However, other engineering disciplines show a big difference in terms of capturing, organising and sharing their design knowledge in order to make routine design simpler.

Second, the maturity of *software measurement*[1] is quite unsatisfactory for software engineering. It is obvious that an improvement in software productivity and quality cannot be assured unless systems are properly measured and, hence, there exists a basis for a better understanding of these two key factors. One major problem with measuring software is that software is intangible unlike other real world objects. Therefore, when measurement is mentioned in the software domain, it actually means measuring the

[1]The terms *software measurement* and *software metrics* themselves are often used interchangeably within the software engineering community, revealing the confusing situation of *software measurement*.

reflection of programs as in the form of source code or other project artifacts like design documents. Another problem with software measurement is the proliferation of software metrics. Many similar kinds of metrics have been proposed without ever attempting to unify or standardise them as in the case of OO methods, e.g, the Unified Modelling Language (UML). Therefore, software engineers often find it difficult to decide which metrics they would use for their specific projects.

As a result, the goal of software engineering has not been fully achieved. Developers are still observing their software projects ending disastrously. Further those delivered systems often do not live up to their original expectations. These events have caused users to adapt themselves to their systems, not the other way around forcing their business activities to be modified undesirably.

These two drawbacks have been partly caused by the fact that software is not tangible, thus it is difficult to manage it in the first place. Further software systems are considered disposable, focussing on the development of one specific application rather than building a family of systems that can be reused later. The research work on "component-based software engineering (CBSE)" [6] and "product line architectures" [53] is related to this problem.

1.2 A Proposed Solution

Software design activities are one of the most time-consuming and creative tasks during the software life cycle, thereby requiring a high degree of human intelligence. This explains the reason why the transition from analysis into design is often figuratively described as the miraculous transformation of a caterpillar into a butterfly, in other words, "metamorphosis". Also, as faults originating from the early stages of the software life cycle need more efforts to fix them than those introduced in the later coding stage, time spent on design can have an impact on later stages [123].

Reuse in general provides a basis for intellectual progress in most human endeavours. While code reuse can save time and effort to some extent, it must be noticed that the savings will obviously not exceed the coding time which is approximately 13% of the whole investment during the software life cycle [59], although this rate varies depending on the types of systems [22]. Much bigger savings can be made from reusing artifacts produced during design, testing, and maintenance. Thus, reusing software designs is considered a good way of improving programmers' productivity and software quality.

The object-oriented programming (OOP) paradigm developed after the structured programming paradigm has been touted as a solution to tackle the software crisis by providing powerful reuse facilities like *inheritance* for static reuse and *polymorphism* for dynamic reuse at the code level. However, its limitations have long been

recognised, and alternative approaches or complementary approaches to overcoming its weaknesses have eagerly been researched for the past two decades [100]. Three of the most representative ones are based on the concepts of *software architecture*, *(object-oriented) frameworks*[2] and *software design patterns* [46, 134, 72, 45].

This thesis attempts to find a way of achieving a high degree of design reuse practised in other mature engineering disciplines by applying design patterns. To assist the OOP paradigm, many object-oriented analysis and design (OOA/D) methods have been developed. They model the real world using various new features; *inheritance* and *encapsulation* are two of their most useful features. However, they fail to describe the overall system structure and have not brought the same degree of extensive design reuse experienced in other engineering disciplines. For example, it is difficult to identify suitable classes and their relationships upon building new systems, thus later ending up refactoring classes into suitably sized ones [93]. It is argued that that patterns can improve these situations as each pattern embodies a collection of classes and their relations.

[2]In the software engineering community, *object-oriented frameworks* or *object-oriented application frameworks* are simply called *frameworks* as these are built under object-oriented programming environments. *Abstract classes* and *object composition* are two of the most heavily used and useful language features to build frameworks.

1.3 Contributions and Their Criteria for Success

First, the manner in which software systems are developed under the OOP paradigm is studied. By doing so, a coherent understanding of the structure of an OO system is obtained. Then, to improve the current reuse practices, a method to detect design patterns instances is developed. The method uses existing software metrics reported in software engineering literature. Three different categories of metrics are selected; these are procedural, structural and object-oriented metrics [37, 31]. Having identified patterns from a system, the system can be redocumented using these patterns; thereby improving program comprehension.

To validate the soundness of this approach, three criteria for the success of the research are set out.

First, the development of a design pattern recovery (DPR) method[3] is investigated. In the research, development of a method to recover new patterns is not attempted. This kind of process is obviously much more difficult than the first one as the structure and behaviour of patterns are not known. Here, identification of the existence of 23 GoF design patterns catalogued in the design pattern book [45] is made. The proposed method should be as accurate as possible so that users can use it with confidence.

[3]In a strict meaning, the word, "detection" will be a more correct one than "recovery" because finding new patterns is excluded from this research. However, the word, "recovery" is used in this thesis as it is a common term in the software engineering community. It is a very similar kind of phenomenon with the usage of "metrics" and "measurement".

Second, having located patterns, users need to apply these patterns to their projects. Among the many potential usages of patterns, the focus of attention is on redocumenting programs. Some of the XML (eXtensible Markup Language) technologies are used to document patterns and redocument software based on identified patterns. This process comprises the pattern-based redocumentation (PBR) method.

Third and finally, this approach needs to be validated through experimentation in a series of case studies. Several systems are chosen for the experiments. Other people have observed the presence of patterns in these systems.

As a whole, the success of the research will be judged by whether the two methods are powerful enough to detect design patterns and allow successful redocumentation to occur.

1.4 The Outline of the Thesis

The overall organisation of the thesis is as follows:

Chapter 2 gives an overview of background related to the research described in this thesis. The important topics to be dealt with in that chapter are design pattern, software metrics and redocumentation methods.

In Chapter 3, the design pattern recovery (DPR) method is presented. An explanation of how semantic and high-level design information captured in a pattern can

be reverse engineered by analysing syntactic and low-level information of programs is given. To obtain syntactic information software measurement techniques are employed and some statistical analyses are applied to the collected data.

Chapter 4 describes the pattern-based redocumentation (PBR) method whereby detected patterns can be applied to improve the current state of documentation. Pattern-based redocumentation is useful for improving program comprehensibility, thus reducing future maintenance costs. The PBR method is realised through utilising XML technologies in order to ensure consistent representation, validation of information, and a high degree of flexibility.

To investigate the usefulness and soundness of this research method, a case study is conducted in the following chapter. The experimental results and their analyses are given.

Finally, in Chapter 6, a summary of the major research contributions is given, and the research results are evaluated. Also, some further work is suggested for future research.

Chapter 2

Background of the Research

This chapter describes the background of this research. First, the main causes of the *software crisis* and some existing solutions for them are discussed. Then, software reuse is suggested as one of the most promising approaches to overcoming this crisis. The limitations of the more traditional code-based reuse methods are pointed out along with arguments for the benefits from reusing higher level artifacts such as designs. Three of the most representative *design reuse* approaches are identified. They are *software architecture*, *object-oriented frameworks*, and *design patterns* [46, 134, 45]. Among these, patterns are chosen as the most promising approach to meet the aim of reusing software components, especially those developed in the object-oriented programming paradigm. Then, an overview of software measurement is given as software product metrics are used to detect patterns from legacy systems in the research. Finally,

the history and development of software documentation is surveyed and a special emphasis is put on XML (eXtensible Markup Language). XML technologies are used to document a pattern catalogue and redocument a system with detected patterns in Chapter 4.

2.1 Introduction

Since the dawn of the modern computing era, i.e., back in 1940s, the huge gap between the development of hardware and software has resulted in the software crisis, represented by low productivity and poor quality of software systems produced. Recognising this kind of urgent phenomena, the term, "software engineering" was coined at the First Software Engineering Conference held in Garmisch, Germany in 1968 [84]. Although there exist many definitions depending on each person's different perspective, a typical definition of software engineering reads [90]:

> The established use of sound engineering principles in order to obtain software economically that is reliable and works efficiently on real machines.

This definition is good in that it addresses the point that software engineering can become a fully fledged engineering discipline only if engineering principles are applied. The issues concerning sound engineering principles have been already discussed in Chapter 1.

A more comprehensive definition of software engineering is:

> Software engineering is the science and art of specifying, designing, implementing and evolving – with economy, timeliness and elegance – programs, documentation and operating procedures whereby computers can be made useful to man [82].

This is a good definition in that it stresses the art, i.e., creativity, required in software engineering, at all stages of the software life cycle, the economics of the software engineering process and the fact that software engineering involves the production of more than just program code.

As with traditional engineering, software engineering involves the use of a rigorous method for software production [28].

> A method is a set of procedures (guidelines) for selecting and sequencing the use of tools and techniques [21].

Therefore it is true to say that in software engineering it is very important to build various models and to develop methods to support and validate them, considering that software is an intangible object.

It is interesting to see that at the First Software Engineering Conference McIlroy proposed the idea of producing software out of prefabricated software components [84]

as a solution to the software crisis. His original intention was that if portions of previously built software, say *software components* are reused, programmers' productivity can be increased and, at the same time, software quality can be improved; as time-tested software components are used. Although some progress has been made to realise his vision into reality, it is still a long way before the same level of reuse as in other engineering disciplines is achieved.

In the next section, the implications of reusing software components are studied in more detail.

2.2 Software Reuse

Looking back to the history of computing it is observed that both in the hardware and software communities developers have been trying to reuse processes, products and resources in order to maximise their ability to cope with ever demanding and changing users' requirements. For example, consider the speed with which RAM (Random Access Memory) and microprocessors are upgraded these days. Moore's Law explains this well. In 1965 Intel co-founder Gordon Moore made an observation while preparing a speech, that each new memory integrated circuit contained roughly twice as much capacity as its predecessor, and each chip was released within 18 to 24 months of the previous chip. If this trend continued, he reasoned, computing power would rise exponentially with time [88, 85].

Date	Chip	Transistors	MIPS	clock/MHz
Nov 1971	4004	2300	0.06	0.108
Apr 1974	8080	6000	0.64	2
Jun 1978	8086	29000	0.75	10
Feb 1982	80286	134000	2.66	12
Oct 1985	386DX	275000	5	16
Apr 1989	80486	1200000	20	25
Mar 1993	Pentium	3100000	112	66
Nov 1995	Pentium Pro	5500000	428	200

Table 2.1: The improvement made on processor chips between 1971 and 1995

Moore's observation still holds today and is the basis for many performance forecasts. In 24 years the number of transistors on processor chips has increased by a factor of almost 2400, from 2300 on the Intel 4004 in 1971 to 5.5 million on the Pentium Pro in 1995, doubling roughly every two years as shown in Table 2.1.

In short, software engineers have the potential to build more powerful software because they have more powerful hardware, but having this potential does not mean that building larger systems is any easier. They need to apply more rigorous engineering principles, e.g., design reuse and measurement, to the development of software-intensive systems as done in the hardware industry.

Coming back to the software domain, there exist many definitions of software reuse.

Although a quite narrow definition that "software reuse is the re-application of source code" is possible, much broader definitions are needed to accommodate reuse approaches at higher abstraction levels and on a broader scale than source code. After all, only 13% of the investment made during the software life cycle is spent in the coding phase [59]. More time and a larger proportion of the budget are spent on maintaining and evolving software rather than developing it, thus forcing users to be concerned about program comprehension to prepare for those activities.

In terms of the above facts, Biggerstaff's following definition of software reuse is more suitable and useful for this investigation [18]:

> Software reuse is the re-application of various types of knowledge about a certain system with the aim of reducing the burden of development and maintenance. The reusable elements consist of domain knowledge, development experiences, project choices, architectural structures, specifications, code, documentation and so on.

According to the above definition, anything produced and used during a software project becomes potentially an object of reuse.

2.2.1 Existing Types of Software Reuse

Many different kinds of reuse have been identified and it is not easy to classify them according to any strict criteria. The reason is that they are often applied in a combined way. It is very rare that only one single method is used. Below existing reuse types that frequently appear in the literature are summarised.

First, reuse can be classified into *systematic* and *non-systematic reuse* according to the degree of how carefully software reuse schemes are planned and managed [106]. Systematic software reuse means:

- understanding how reuse can contribute toward the goals of the whole business,
- defining a technical and management strategy to achieve maximum value from reuse,
- integrating reuse into the total software process, and into the software process improvement programme,
- ensuring all software staff have the necessary competence and motivation,
- establishing appropriate organisational, technical and budgetary support, and
- using appropriate measurements to control reuse performance.

Non-systematic reuse is, by contrast, ad hoc, dependent on individual knowledge and initiative, not deployed consistently throughout the organisation, and subject to

little if any management planning and control. If the parent software organisation is reasonably mature and well managed, it is not impossible for non-systematic reuse to achieve some good results. The more probable outcome, however, is that non-systematic reuse is chaotic in its effects, feeds that high risk culture of individual heroics and fire-fighting, and amplifies problems and defects rather than dampening them.

Next, reuse can be classified by the artifacts that are reused [83]. In essence, every artifact produced during the software life cycle can become an object for various reuse methods, including requirements, specifications, designs, code, documentation, and test cases. It is not unusual that "defined processes" are included into this category.

One of the most popular classification schemes is judging by the degree of modifications made before reuse takes place [7, 17, 106]. If an asset is reused without the need for any adaptation, this is known as *black box reuse*. If reengineering is necessary, that is to say if it is necessary to change the internal body of an asset in order to obtain the required properties; this is the case of *white box reuse*. The intermediate situation, where adaptation is achieved by setting parameters, is called *grey box reuse*. *Glass box reuse* refers to the situation where it is necessary to "look inside" an asset, on a "read-only" basis, in order to discover its properties, in the case where the available description of those properties is inadequate. It has long been believed that non-modification style reuse, i.e., black box reuse, is the most desirable. However, the lack of adequate technology has hindered software engineers from achieving this.

Recently, *component-oriented programming (COP)* technologies such as *binary composition techniques* represented by Microsoft's DCOM (Distributed Common Object Model) and OMG (Object Management Group)'s CORBA (Common Object Request Broker Architecture) are starting to enable developers to reuse without requiring modifications [133, 116, 94]. While DCOM is only for the Microsoft Windows platforms, CORBA is basically platform independent. These new technologies certainly opened a new horizon for the success in reuse. However, it will be still a long way before COP is a main stream technology like OOP.

The scope of domains where reuse is achieved can be used to divide reuse into two groups, i.e., *vertical reuse* and *horizontal reuse* [106]. In general, the term, vertical reuse is used to refer to reuse which exploits functional similarities in a single application domain. It is contrasted with horizontal reuse, which exploits similarities across two or more application domains. There are two forms of horizontal reuse. The first refers to the exploitation of functional similarities across different domains; an example might be loans and reservations functions in the domains of libraries and car hire. The second refers to the exploitation of similarities in technical domains such as user interface and operational platform, which are independent of application domains. It is generally believed that vertical reuse is easier to achieve than horizontal reuse. Thus greater emphasis is placed on it. Evidence of this belief is research on domain-specific software architectures (DSSAs) [47].

Finally, Biggs argues that different approaches should be adopted depending on

the size and available resources of the organisations where a reuse scheme can take place [20]. In short, very often reuse approaches that work well in big organisations are not directly applicable to small ones without some sorts of adaptation.

2.2.2 Factors Militating against Software Reuse

Since McIlroy suggested the idea of building software out of prebuilt software components, software reuse has been a dream of many software engineers [84]. However, this dream is yet to be fully realised. There are many reasons why software engineers have failed to realise the potentials of reuse, which include both technical and non-technical issues.

There are four kinds of barriers that have to be tackled before widespread reuse can be realised. They are technical, cultural, managerial and legal factors. It has been reported that non-technical aspects are as important as technical ones [122, 107].

Technical Factors

Sommerville identified six technical problems to be solved for the success of software reuse [124].

First, desirable attributes for reuse are to be investigated. Once the characteristic, *reusability* is known, highly reusable, new components can be developed. Also, existing components can be reengineered in a cost-effective way to increase their reusabil-

ity [62]

Second, methodology problems arise since most existing software design methods are intended to support software development without reuse. Therefore a new development methodology is needed to open the so-called "software component industry". In other words, there is a consensus that *design-for-reuse* should precede *design-with-reuse* [63].

Third, new documentation standards for reusable components need to be established. The documentation of a reusable component must specify both its functional and non-functional characteristics. Usually, more documentation is required for reusable components than for components which are simply part of a larger system. Ideally, reusable components would be formally specified so that there is no ambiguity about their behaviour. However, this is unlikely to happen in the foreseeable future since formal methods are not fully integrated into standard software development. Thus, more rigid documentation standards should be used to help users reuse their components more easily.

The fourth problem is about how components can be certified as reusable. In order to convince managers of the value of reuse, they must have confidence in the components that will be reused. This implies a need for some kind of *component certification scheme* which will certify the quality or usefulness of the component [140]. But setting up such a scheme has been shown to be both difficult and expensive.

Fifth and probably the most important and frequently mentioned problems in the reuse research community are about component retrieval [54, 87]. In a large company such as an aerospace company there might be potentially hundreds of, if not thousands of, reusable components available. They are collected from many different types of hardware and software projects. Therefore finding what components exist and retrieving these components could be a major problem. A cataloguing scheme using existing database systems must be established.

Finally, configuration management (CM) needs to be carefully planned in the reuse environment [67]. The normal model of configuration management is currently project-based. The software developed as part of a project is maintained in a project archive. On the contrary, reuse requires software to be shared and, perhaps, components to be modified and stored in a software library or a software repository [71, 99]. The following questions associated with configuration management need to be answered for the success of reuse. What relationships should be maintained between the reuse library and the original base components in the CM system? How should changes be propagated? How can traceability back to the original components be implemented? The answers to these questions are still being studied.

Cultural Factors

One of the fundamental questions that has to be answered is whether the structure of a society has an effect on the acceptance of reuse. It has been claimed that there

is a paradox between the application of a software reuse technology and the approach to life in a Western society [123]. In the West, society tends to be very individualistic, with competitiveness rife in almost all fields of life. This results in an innovative approach to product development. It is argued that this conflicts with a reuse technology which relies on cooperation and trust for its successful application. It is noticeable that one of the best examples of success in applying reuse has occurred in "Japanese Software Factories", in a society where a cooperative and paternalistic ethos is supported [79]. The adoption of the SIGMA project by major industrial and academic bodies in Japan is a venture that one would never expect to be undertaken in the West [1].

There is a very widespread phenomenon called "Not-Invented-Here (NIH)" syndrome within the software community. This arises from the fact that software engineering is perceived as a skilled profession, and reuse implies a form of de-skilling, thus there is a lack of motivation to cultivate a reuse technology [48]. This can only be removed by supplying cheap components of high quality and encouraging sufficient management motivation.

Managerial Factors

A major factor in the successful implementation of reuse is its acceptance and encouragement by management [49]. Unless such backing is forthcoming, reuse stands little chance of success. There are many obstacles which have to be reconciled with

the potential benefits from reuse.

The first fact to be taken into consideration is the greater cost of producing reusable code compared to "solution-specific" production [48]. It is not easy to produce "general" or "generic" components that are suitable for reuse. This results in much more time and effort on the part of a software team, and greater cost for the project as a whole. Since project managers are rewarded for producing systems to deadline and within budgetary constraints, there is little incentive for them to encourage the production of generic components.

There is little quantitative evidence of the successful application of reuse in many fields [101, 44]. In incorporation of a reuse technology, management must be prepared to sacrifice short-term returns to gain unquantifiable benefits in the long-term. This is something many organisations are unprepared to risk. The only way this problem is likely to be alleviated is by wider scale availability of component libraries.

Management obstacles to reuse may be the most intractable of all to surmount. The adoption of risk-taking policies is necessary to promote the application of reuse, and demonstrate the immense benefits that can accrue from it. It is very much a "chicken-or-the-egg" situation, requiring enterprising firms who are prepared to sacrifice returns in the short-term for the undeniable but unquantifiable benefits in the long-term.

Legal Factors

There exist two kind of legal issues associated with reuse, i.e. *intellectual property*

right and *liability* [113]. The former forces responsibility to keep copyright, patent, and trade secret laws, whereas the latter is about handling any damage caused by a certain piece of software. Many decisions about the development, distribution, maintenance, enhancement and, especially, reuse of software are likely to be affected by constraints imposed by intellectual property laws and liability laws.

The primary purpose of the intellectual property laws is to encourage the development and dissemination of innovative works for use by the public [112]. The creation or invention of useful items and artistic works generally requires the investment of considerable time, energy, and resources by skilled, talented people. To encourage such activities, the intellectual property laws provide, as an incentive, the opportunity to obtain exclusive rights to commercial exploitation of the innovative or artistic work for a specified period of time. Generally it is said that developing reusable components needs a big initial investment. Thus the developers' rights must be protected. Otherwise reuse would not happen [33]. This results in the fact that active reuse more likely occur within an organisation rather than across different organisations.

Copyright issues arise not only in external reuse environment, but also in internal reuse. For instance, if a component is developed by an employee, who will own its copyright between him and his employer? As another problem, nowadays many components are reverse-engineered. In this case, it must be made sure whether reverse-engineering old legacy code is legal or not.

Another thing that should be considered when software is reused is software product liability. It has been believed that software defects are rarely lethal and the number of injuries and deaths is very small. Software, however, is now the principal controlling element in many industrial and consumer products. Thus, users are starting to realise that software, particularly poor quality software, can cause products to do strange and even terrifying things. Software bugs are erroneous instructions and, when computers encounter them, they do precisely what the defects instruct. As a worst case, an error could cause a 0 to be read as a 1, or, in the case of a radiation machine in a hospital, a shield to be removed instead of inserted. A software error could mean life or death.

The best way to overcome this problem is to develop software of high quality. Software reuse and SEI (Software Engineering Institute)'s CMM (Capability Maturity Model) are such attempts to achieve this goal [96, 97]. But until it becomes common practice, software products liability laws are needed.

The intellectual property laws and liability laws are in the process of evolving to provide adequate and appropriate protection for software. However, there are many questions about these laws for which there are as yet no clear answers [29].

Because this research aims at reusing design information by applying reverse engineering and redocumentation techniques, the above points are very much relevant.

2.2.3 Object-Oriented Programming (OOP) and the Unified Modelling Language (UML)

One of the most popular buzz words for the past two decades or so is *object-oriented programming (OOP)*. The advocates of OOP claim that software reuse is all but automatically guaranteed if software systems are built using OOP. Also many people think that OOP is equal to programming in the C++ Language because of the huge popularity it has gained since its invention. However, OOP is not achieved by simply writing C programs that can be compiled with C++ compilers. There is more to OOP than that. The power of OOP can be extended to much bigger objects, e.g., components and OO frameworks [100].

OOP was the first attempt to achieve code-level reuse by embedding reusability facilities such as *encapsulation* and *inheritance* directly into programming languages themselves, and, hopefully, some degree of design reuse, as well, in the corresponding object-oriented designs [130].

Although there are still many limitations of OOP, generally speaking, OOP has improved software quality and programmers' productivity. These days, people cannot imagine developing large systems without some degree of OOP features. As a typical example, Microsoft Visual BASIC has many OOP features in it as a way of overcoming the weak aspects of the traditional procedural BASIC language.

Realising the difficulties experienced during the early stages of software life cycle,

e.g., requirements, specification, analysis, and design, many object-oriented analysis and design (OOA/D) methods have emerged, provoking the so-called "OO method war" [24]. Some representative ones are Booch's method [23], Rumbaugh's OMT (Object Modelling Technique) [111], and Jacobson's OOSE (Object-Oriented Software Engineering) [56], to name but a few[1]. However, recently the OOA/D notations associated with these methods were incorporated into and standardised as *the Unified Modelling Language (UML)* by the Object Management Group (OMG) [92]. UML is a language for visualising, specifying, constructing, and documenting the artifacts of a software-intensive system.

In UML, five different kinds of views are used to describe a software-intensive system. They are *use case view*, *design view*, *process view*, *implementation view*, and *deployment view* [24]. In addition, these five views are captured and represented through nine different diagrams. They are *class diagram*, *object diagram*, *component diagram*, *deployment diagram*, *use case diagram*, *sequence diagram*, *collaboration diagram*, *statechart diagram*, and *activity diagram*. The first four diagrams are useful for describing the static parts of a system whereas the latter five are used to view the dynamic parts of a system [24].

On top of the above features, UML provides a formal definition of a common object analysis and design (OOA&D) metamodel to represent the semantics of OOA&D models, which include static models, behavioural models, usage models, and archi-

[1] A comprehensive survey of object-oriented methods can be found in Biggs [19].

tectural models [92]. For example, Figure 2.1 shows how various modelling elements are connected together to comprise a class. According to the diagram, a class can have structural feature and behavioural feature, and they all are subclasses of the class, "ModelElement", and so on. It is interesting to see that UML semantics itself is described by using UML.

UML facilitates reuse in terms of both products and processes, and provides users with visual modelling techniques, something like blueprints used in building architecture domain. Although some naive people admire UML as perfect in a similar manner that they did to "OOP" and "design pattern", it has also some limitations. For example, when it was first released, it lacked the capacity of specifying various kinds of constraint on modelling elements. Later OCL (Object Constraint Language) strengthened this weakness of UML [142].

UML is a good conceptual tool for software developers because it is a standard and is based on earlier accomplishments in OOP. As people are collaborating to improve it further, its continued use in the future is certain.

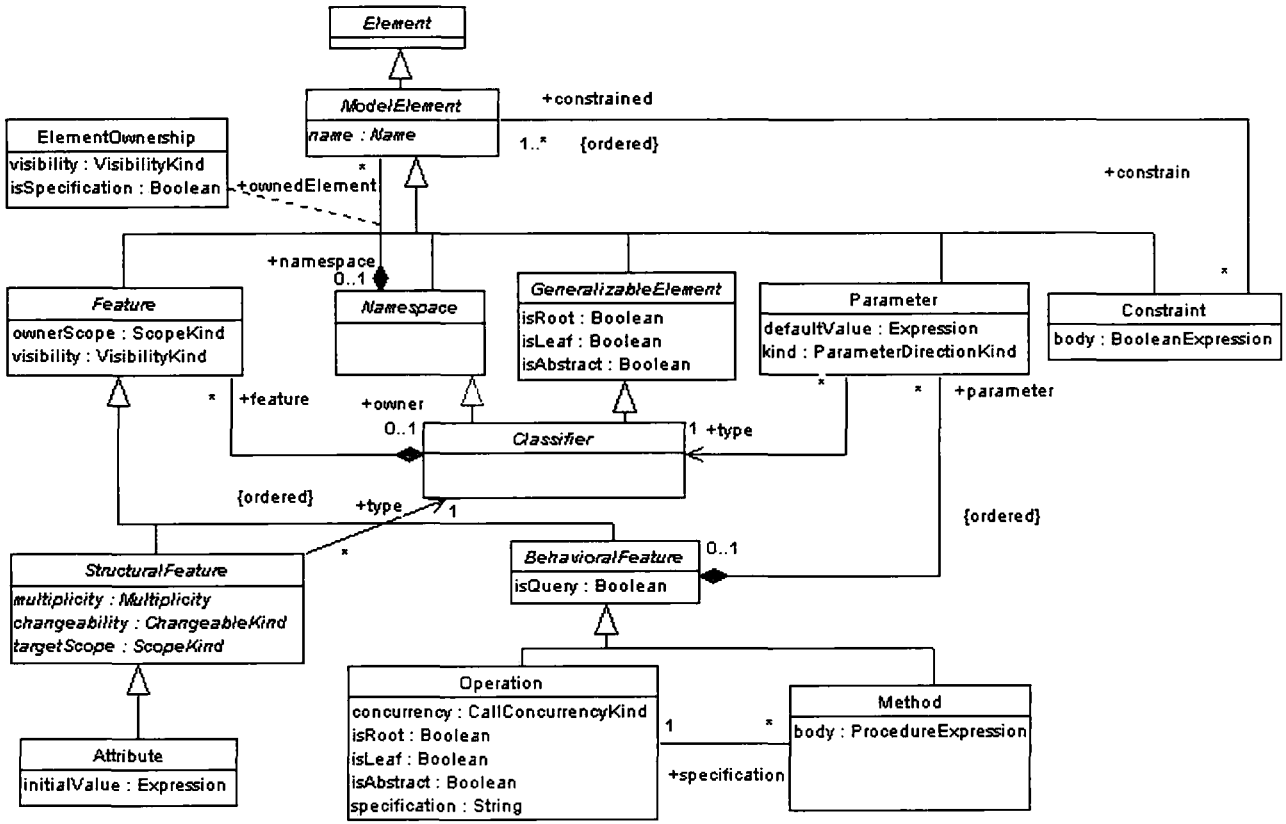

Figure 2.1: UML core package - backbone (OMG UML Specification 1.3)

2.2.4 Software Design Reuse

Influenced by OOP and attempting to overcome its weakness, many promising design reuse approaches have emerged. Although these approaches vary greatly in their methods and scales, they all have one commonality, i.e., they all attempt to capture and use higher abstractions than those more traditional small scale ones such as data structures and algorithms [83].

As explained Chapter 1, design reuse can bring much greater benefits than simply reusing code. The idea of reusing software designs is not new. One of the earliest examples of design reuse was the DRACO approach proposed by Neighbors at the University of California at Irvine in the early 1980s [91]. He attempted to construct software systems from reusable software parts. In particular he was concerned with the reuse of analysis and design information in addition to programming language code. The goal of his work on DRACO was to increase the productivity of software specialists in the construction of similar systems. The particular approach he took was to organise reusable software components by problem area or domain. Statements of programs in these specialised domains are then optimised by source-to-source program transformations and refined into other domains. However, at that time he could not implement his idea fully because of the inadequate technology.

Based on Neighbors's work and improving it further, Batory has been working on *software generators* at the University of Texas at Austin [14]. He argues that the pro-

duction of well-understood software will eventually be the responsibility of software generators and generators will enable high-performance, customised software systems and subsystems to be assembled quickly and cheaply from component libraries. These components are intelligent and they encapsulate domain-specific knowledge, e.g., best practitioners' approaches, so that their instances will automatically customise and optimise themselves to the system in which they are being used. Currently Batory et al. are transferring their technological innovations to Microsoft in order to help the company's software production lines like its popular Office suite.

Since the broad adoption of OOP, people have been eager to develop large-scale, higher abstraction-based reuse approaches. Two of the most prominent ones are *object-oriented frameworks* and *design patterns* [72, 45]. These depend on each other very much as observed by many researchers both from the industry and the academia. Evidence for this fact is that most GoF patterns were discovered by developing OO frameworks and then later reflecting on them.

A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems [45, 5]. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context.

Framework is a set of cooperating classes that makes up a reusable design for a specific class of software [57, 72]. A framework provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations. A developer customises the framework to a particular application by subclassing and composing instances of framework classes.

Originally independent from these two approaches, research on *software architecture* has been carried out, mainly identifying *architectural styles* and building *domain-specific software architectures (DSSAs)* [137]. The Software Engineering Institute (SEI) sponsored by the Department of Defense is the champion of this area of research. This is probably the largest scale design reuse approach at present [47, 118].

In terms of abstraction level, software architecture and design patterns are at a higher level than frameworks. This is explained by the fact that the former two are not necessarily linked to any implementation details while the latter is tightly associated with implementation details through OOP features such as *abstract classes* and the *object composition* mechanism.

In the mean time, with respect to the size, the order is a little bit different, software architecture being the largest, design patterns being the smallest, and frameworks positioned in the middle between them. It is true that frameworks can contain many patterns, while the reverse is never so. Also an instance of software architecture is often made of several frameworks.

Another noticeable thing about these three design reuse approaches is that frameworks and patterns are realised in OOP, whereas software architecture covers much broader areas beyond OOP.

Because only the design pattern concept is used in this research, an in-depth study on it is carried out in the following section.

2.3 Design Patterns

2.3.1 Definition

Software design patterns are an emerging concept for guiding and documenting system design. The original interest in patterns was sparked by the work of an architect, Christopher Alexander, whose patterns encode knowledge of the design and construction of communities and buildings [3, 2]. His use of the word "pattern" takes on more meaning than the usual dictionary definition. Alexander's patterns are both a "description" of a recurring pattern of architectural elements and a "rule" for how and when to create that pattern [35, 15]. They are the recurring decisions made by experts, written so that those less skilled can use them. They describe more of the "why" of design than a simple description of a set of relationships between objects.

It is remarkable that one of most popular set of design patterns, so-called Gang of

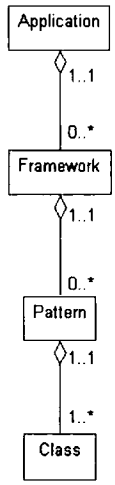

Figure 2.2: Relationships between different software components

Four (GoF) patterns[2] were discovered while developing frameworks such as ET++ and HotDraw and reflecting on them later [45]. This shows good quality applications and frameworks do contain many kinds of patterns. Figure 2.2 shows the relationships between class, pattern, framework, and application using the UML class notation. Patterns are constructed of classes and/or objects. In turn, they comprise frameworks. Finally, application can be instantiated from existing frameworks. The containment existing between these software components are expressed using "aggregation" and "mutiplicity". For example, a pattern contains at least one class by its own nature, whereas a framework might not include any patterns at all, although it is desirable.

[2]In the pattern community, Gang of Four indicates the four authors who wrote the book, "Design Patterns: Elements of Reusable Object-Oriented Software", i.e., Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides.

2.3.2 Describing Design Patterns

There have been many attempts to specify patterns more precisely either using formal specification or graphical notations [69, 70]. However, it does not appear that a high degree of success has been achieved. This is mainly because the approaches are important and useful but they are not sufficient; they simply capture the end product of the design process as relationships between classes and objects. To reuse the design, there is a need for recording the decisions, alternatives, and trade-offs that led to it [45].

In general, a pattern has four essential elements, i.e., **pattern name**, **problem**, **solution**, and **consequences** [45].

First, the **pattern name** is a handle that describes a design problem, its solutions, and consequences in a word or two. Like identifiers used in programs, choosing suitable names for patterns is very important because they are important media of communication between developers. For example, in the structural programming paradigm, if "stack" is mentioned; then the audience can be immediately reminded of the details of the data structure. In a same token, by naming patterns properly, quite a large chunk of information can be conveyed to others without explaining the details.

Second, the **problem** describes when to apply the pattern.

Third, the **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations.

Finally, the **consequences** are the results and trade-offs of applying the pattern. The consequences resulting from one pattern becomes a starting point from which other patterns can be applied. This is where pattern languages move in. Pattern language encapsulate a collection of patterns that tend to collaborate to achieve bigger goals that one individual pattern can [16].

Containing the above four essential elements, the so-called GoF Pattern Template is one of the most popular way of describing individual patterns. It was adapted from Alexander's Pattern Template by the GoF [3]. The GoF Pattern Template has 13 sections to describe each pattern [45]:

Pattern Name and Classification: The pattern's name conveys the essence of the pattern succinctly. A good name is vital, because it will become part of design vocabulary.

Design patterns can be classified by two criteria as shown in Table 2.2. The first criterion, called "purpose", reflects what a pattern does. Patterns can have either creational, structural, or behavioural purpose. Creational patterns concern the process of object creation. Structural patterns deal with the composition of classes or objects. Behavioural patterns characterise the ways in which classes or objects interact and distribute responsibility.

The second criterion, called "scope", specifies whether the pattern applies primarily to classes or to objects. Class patterns deal with relationships between classes and their subclasses. These relationships are established through inheritance, so they are static,

		Purpose		
		Creational	**Structural**	**Behavioural**
Scope	**Class**	Factory Method	Adapter(class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter(Object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Table 2.2: Design pattern categories (Gamma, 1994)

i.e., fixed at compile-time. Object patterns deal with object relationships, which can be changed at run-time and are more dynamic. Almost all patterns use inheritance to some extent. So the only patterns labeled "class patterns" are those that focus on class relationships. Most patterns are in the object scope.

Creational class patterns defer some part of object creation to subclasses, while creational object patterns defer it to another object. The structural class patterns use inheritance to compose classes, while the structural object patterns describe ways to assemble objects. The behavioural class patterns use inheritance to describe algorithms and flow of control, whereas the behavioural object patterns describe how a group of objects cooperate to perform a task that no single object can carry out alone.

Intent: The intent of a pattern describes the rationale for using the pattern.

Also Known As: This section indicates other well-known names for the pattern, if any.

Motivation: A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help users understand the more abstract description of the pattern that follows.

Applicability: This section explains the situations in which the pattern can be applied.

Structure: A graphical representation of the classes in the pattern using a notation

based on the Object Modeling Technique (OMT). Interaction diagrams are also used to illustrate sequences of requests and collaborations between objects. Since the emergence of UML, people use it to describe the structure of a pattern instead of using other various OO notations. Obviously, by using the standardised UML the communication between developers can be improved.

Participants: The classes and/or objects participating in the pattern and their responsibilities.

Collaborations: This section addresses the collaborations between the participants of the pattern to carry out their responsibilities.

Consequences: The trade-offs and results of using the pattern are dealt with in this section.

Implementation: The details of implementing the pattern are described.

Sample Code: Code fragments that illustrate how the pattern might be implemented in an OO programming language.

Known Uses: Examples of the pattern found in real systems are shown.

Related Patterns: This sections describes which other patterns are closely related to this one, and the differences between them.

The GoF pattern catalogue contains 23 design patterns. Below a short description of each pattern is given based on their names and intents in alphabetical order [45].

1. Abstract Factory: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

2. Adapter: Convert the interface of a class into another interface that clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.

3. Bridge: Decouple an abstraction from its implementation so that the two can vary independently.

4. Builder: Separate the construction of a complex object from its representation so that the same construction process can create different representations.

5. Chain of Responsibility: Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

6. Command: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

7. Composite: Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

8. Decorator: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

9. Facade: Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

10. Factory Method: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

11. Flyweight: Use sharing to support large numbers of fine-grained objects efficiently.

12. Interpreter: Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

13. Iterator: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

14. Mediator: Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

15. Memento: Without violating encapsulation, capture and externalise an object's internal state so that the object can be restored to this state later.

16. Observer: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

17. Prototype: Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

18. Proxy: Provide a surrogate or placeholder for another object to control access to it.

19. Singleton: Ensure a class only has one instance, and provide a global point of access to it.

20. State: Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.

21. Strategy: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

22. Template Method: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

23. Visitor: Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

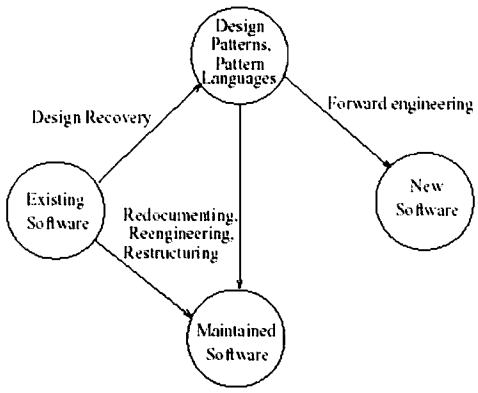

Figure 2.3: Software engineering processes related to design patterns

2.3.3 Using Design Patterns

Five kinds of software engineering processes are associated with design patterns as explained in Figure 2.3. They are *design recovery process*, *redocumentation process*, *restructuring process*, *reengineering process*, and *forward-engineering process* [57, 115, 102, 114, 129, 60]. The first one focusses on recovering design patterns, while the other four are about the usage of those recovered patterns or the usage of known patterns. First of all, it is necessary to recover design patterns from existing software by applying design recovery techniques. There exist two kinds of design pattern recovery. One is detecting the existence of already known patterns like GoF patterns. Another is finding new patterns. Certainly the latter process will be much more difficult than the former. In a sense, this is similar to data mining in artificial intelligence. Having identified and catalogued them in pattern repositories, those patterns can be applied to redocument software for improving program comprehensibility, and to restructure it

into a more desirable shape in terms of those specific software quality issues that users are interested in such as low coupling and high cohesion. Also some users might want to reengineer their software to adapt to new technologies like Enterprise Java Beans (EJBs) and CORBA, or to meet the requirements of new business environments like e-commerce. Finally, those patterns can be used in the development of new software, i.e., forward engineering process, along with the usual data structures like *stack* and *queue*.

2.3.4 Existing Work on Design Pattern Recovery

Fundamental to all pattern investigations is the attempt to recognise recurring situations in design so that users can learn from other people's experience. The processes that investigators use to find patterns vary widely. There are three general categories [61].

The *introspective approach* is when people reflect on the systems that they have built and find patterns relating to their experience. This approach can be described as a search for individual architectural style. The work of Shull et al carried out at University of Maryland belongs to this category [119].

The *artifactual approach* studies systems built by different teams working on similar problems. The pattern investigator is not involved with system development and seeks a more objective perspective. This approach can be described as a study of the software

artifacts. Most work including the one developed in this thesis follows this approach.

The *sociological approach* studies the people building similar systems to discover the recurring problems in the systems and in developer interactions. This technique can best be described as an investigation through interview. Patterns that are recovered from this approach tend to be at higher abstractions such as requirements and analysis patterns [43].

Patterns thinking is new; so, too, is patterns investigation. No doubt there are other approaches, and some researchers are using a combination of methods.

A few pieces of work on discovering design patterns from existing applications have been reported from academia and industry [78, 27, 64]. However, most of the results are special cases or, if general, inefficient for applying to industrial applications. Further, some ways of identifying patterns are language-specific so that people cannot use those methods for applications built with other programming languages. A typical one is the work done with programs in Smalltalk by Brown [27].

2.4 Software Measurement

Effective management of any process requires quantification, measurement, and modeling. Software metrics provide a quantitative basis for the development and validation of models of the software development process [37]. Metrics can be used to

improve software productivity and quality. Below some of the most commonly used software metrics are introduced and their use in constructing models of the software development process is reviewed. Although current metrics and models are certainly inadequate, a number of organizations are achieving promising results through their use. Results should improve further as additional experience with various metrics and models is gained.

Software metrics are necessary to know the properties of the software that are developed and predict the needed effort and development period. Moreover, they are needed when software is maintained for various reasons that fall into the four kinds of maintenance, i.e *corrective maintenance*, *adaptive maintenance*, *perfective maintenance* and *preventive maintenance* [132].

The history of measurement is as old as human history. Among those measuring units, some such as *foot* still exist until now. It is believed that one of the most important concepts in engineering discipline is measurement [42], as is *reuse*. An engineer needs to know why to make measurements, what can be measured, how to measure, and what to do with the results.

Confusion in using terms such as *metrics* and *measurement* proves that the area is still a young discipline, and has been neglected by computer scientists. Lorenz defines the terms as follows [76]. *Metrics* is a standard of measurement used to judge the attributes of something being measured, such as quality or complexity, in an objective

manner. On the other hand, *measurement* is the determination of the value of a metric for a particular object. Therefore, considering these two definitions, the term, *measurement* should be used, when mentioning the activity itself to measure something. However, since the term, metrics, is generally accepted and used in the discipline of software engineering, the distinction between two terms is not strictly made in this thesis.

2.4.1 The Goal/Question/Metric (GQM) Method

Originally proposed by Basili at the University of Maryland at College Park, the GQM method is based upon the assumption that for an organisation to measure in a purposeful way it must first specify the goals for itself and its projects, then it must trace those goals to the data that are intended to define those goals operationally, and finally provide a framework for interpreting the data with respect to the stated goals [10, 8]. Thus it is important to make clear, at least in general terms, what informational needs the organisation has, so that these needs for information can be quantified whenever possible, and the quantified information can be analysed in order to determine whether or not the goals have been achieved.

The approach was originally defined for evaluating defects for a set of projects in the NASA Goddard Space Flight Centre environment [12, 13]. The application involved a set of case study experiments and was expanded to include various types of experi-

mental approaches. Although the approach was originally used to define and evaluate goals for a particular project in a particular environment, its use has been expanded to a larger context. It is used as the goal setting step in an evolutionary quality improvement paradigm tailored for a software development organisation. The result of the application of the Goal Question Metric method is the specification of a measurement system targeting a particular set of issues and a set of rules for the interpretation of the measurement data. The resulting measurement model has three levels [9]:

1. Conceptual level (Goal): A goal is defined for an object, for a variety of reasons, with respect to various models of quality, from various points of view, relative to a particular environment. Objects of measurement can be divided into "products", "processes", and "resources".

2. Operational level (Question): A set of questions is used to characterise the way the assessment/achievement of a specific goal is going to be performed based on some characterising model. Questions try to characterise the object of measurement (product, process, resource) with respect to a selected quality issue and to determine its quality from the selected viewpoint.

3. Quantitative level (Metric): A set of data is associated with every question in order to answer it in a quantitative way. The data can be either objective or subjective. For example, "Lines of Code (LOC)" is an objective metric whereas "maintainability" is a typical subjective metric. It has been known that subjective

metrics are much more difficult to obtain than objective ones as the former ones depend on both the objects that are being measured and the viewpoints from which they are taken.

2.4.2 Kinds of Software Metrics

There are many ways of classifying software metrics. One popular classification scheme is based on the objects of measurement, i.e, process metrics and product metrics. Process metrics try to improve software quality and productivity by measuring properties of the processes of developing software systems whereas product metrics are used to measure the characteristics of each software product or component [37].

As for software product metrics, they can be classified by many different criteria. One way is to divide product metrics according to the programming paradigms in which the subject system is developed. Thus, *procedural*, *structural*, and *object-oriented metrics* exist [55, 42, 31, 76]. Procedural metrics measure properties of software parts such as sizes of each module whereas structural metrics are based on the relationships of each module with others. Other names cited for these two kinds of metrics in the software metrics literature are *intra-module metrics* and *inter-module metrics*, respectively. For example, "lines of code (LOC)" and "McCabe's cyclomatic number (MVG)" are two of the most representative procedural metrics while "coupling" and "cohesion" belong to the structural metrics group [81, 34]. On top of these,

another kind of metrics, i.e., OO metrics have to be considered. Among many OO metrics proposed, Chidamber and Kemerer's OO metrics suite, in short, CK class metrics, is the most popular [31]. They proposed six new class metrics. They are "weighted methods per class (WMC)", "depth of inheritance tree (DIT)", "number of children (NOC)", "coupling between object classes (CBO)", "response for a class (RFC)", and, finally, "lack of cohesion in methods (LCOM)". With respect to the scope covered by each group of metrics, object-oriented metrics are a superset of the other two, because object-oriented systems contain those features that can be found in the traditional programming concepts as well as newly added ones; whereas the the reverse is never true. Thus, specific attributes of OO systems are not reflected well using only procedural metrics and structural ones.

2.5 Software Documentation and Redocumentation

To manage documents produced during software development, many documentation methods have been developed. Here four representative ones are discussed. They are the *literate programming*, *hypertext-based redocumentation method*, *object-oriented documentation method*, and, finally, *pattern documentation method.*

Literate programming was originally proposed by Knuth, and it is a programming methodology that combines a *programming language* with a *documentation language*, making programs more robust, more portable, and more easily maintained than pro-

grams written only in a high-level language [65]. The main idea is treat a program as a piece of literature, addressed to human beings rather than to a computer. The program is also viewed as a hypertext document as with the World Wide Web (WWW). Users combine the use of a text formatting language such as TeX, which is another invention by Knuth, and a conventional programming language so as to maintain documentation and source together. The program is sometimes marked to distinguish it from the text, rather than the other way around as in normal programs. CWEB is an example of one of many working literate programming prototypes.

Fletton and Munro developed a method to redocument software systems using hypertext technology [38, 39]. They argue that software documentation should be produced as a by-product of the development process and handed over as a complete package along with the source code to the team that will maintain the program. Their method is a hypertext-based technique for browsing and documenting source code using a purpose-built prototype. The system uses hypertext links to allow programmers to locate areas of interest rapidly and efficiently in source code and to examine and update related documentation.

Matthews and Grove developed a documentation method based on object-oriented concepts [80]. They proposed that the principles of object-oriented design, originally developed to address software complexity, can also be applied to documentation.

There have been a few pieces of work on explicitly using patterns for documentation

and redocumentation purposes, although this work has by no means been complete. Among them, Prechelt and Unger performed some experiments where they put pattern information into programs as internal documentation, i.e., comments [104, 103]. They found that documenting design patterns in code as internal documentation eases program maintenance.

Finally, Johnson did some research on documenting frameworks using patterns [57]. He argues that the documentation for a framework must meet several requirements to encourage its use. He attempts to meet these requirements by structuring the documentation as a set of patterns, sometimes called a *pattern language*. He claims that patterns can describe the purpose of a framework, can let application programmers use a framework without having to understand in detail how it works and can teach many of the design details embodied in the framework.

2.5.1 Document Management: XML

The origin of XML (eXtended Markup Language) can be traced to SGML (Standard Generalised Markup Language) [51]. The intention of the invention of the language was efficient document management. HTML (Hypertext Markup Language), a more immediate descendent of SGML has done much to facilitate Internet revolution for the past decade. However, the tags available in HTML are limited, and as companies like Netscape or Microsoft added their own tags to HTML, users are gradually fac-

ing incompatibility problems. Thus the need for more open and extensible markup languages has arisen [146].

Some recent development on XML technologies are studied below.

An XML document is well-formed only if it conforms to basic rules of XML such as [52]:

- It must have start and end tags for every element.
- It must have one, and only one, root element.
- Empty elements are formatted correctly.
- The case of start and end tags can be either uppercase or lowercase, but they must match.
- Elements must nest correctly.
- Attribute values must always be in quotes.

A valid document is well-formed and has been validated against a DTD (Document Type Definition) or other specified XML Schema. This means that the document conforms to the rules of the DTD or Schema associated with the document.

A DTD describes the grammar expected of documents that use its vocabulary. Just as English grammar helps writers form proper sentence structure, XML grammar helps authors create properly structured documents. The expected XML grammar is

recorded in the DTD. The DTD provides the means to verify the document's conformance: that is, its validity.

XML schema emerged as a way of overcoming some limitations of DTD. For example, DTDs are very limited in their descriptive powers because they are based on the use of EBNF syntax.

XSL (eXtensible Style Language) is a very powerful tool for transforming XML documents into other formats by transforming an XML document into a separate tree structure. Currently, XSL is used primarily to transform XML semantics into a display format, such as the kind of display used in Web browsers. Despite considerable debate about semantics within the XML/XSL development community, XSL has moved along rapidly as a viable XML presentation language.

HTML linking only goes in one direction, i.e., transporting viewers from one page to another. Because of this limitation, the XML community have been trying to extend the linking facilities to be used with XML documents. XML linking and addressing mechanisms are specified in three W3C (World Wide Web Consortium) Working Draft Documents. These are XML Path Language (XPath), XML Pointer Language (XPointer), and XML Linking Language (XLink) Draft Documents. These are briefly explained below.

The primary purpose of XPath is to do the actual addressing of parts rather than the whole document. The name XPath comes from "path notation", which is used for

navigating through the hierarchical structure of an XML document.

XLink uses XML syntax to create structures to describe both the simple unidirectional hyperlinks of today's HTML as well as more sophisticated multi-ended and typed links. The important part of XLink is that it defines the relationship between two or more data objects as opposed to a whole document.

XPointer builds on XPath to support addressing into the internal structures of XML documents. Thus, it is possible to use the XML markup to link to specific parts of another document without supplying an ID reference.

Tool builders are working rapidly to exploit these new technologies. At the time of writing this thesis, only few web browsers support a certain degree of XML functionality.

In Chapter 4, the PBR method uses some of XML technologies to document a catalogue of patterns and redocument a system using detected patterns.

2.6 Summary

This chapter gave the background information on the research presented in this thesis. First the emergence of the software engineering discipline as a solution to the software crisis was reviewed. This is followed by a discussion on reuse in general and design reuse in particular. Also the importance of measuring software as a means of

improving its quality and predicting its certain characteristics has also been mentioned. Finally, a survey of the recent developments on XML and its related technologies was conducted. This chapter concludes that XML helps users exchange their data more easily and effectively.

Chapter 3

The Design Pattern Recovery (DPR) Method

To apply patterns during software maintenance and evolution, it is essential to detect patterns from legacy systems. This chapter shows a method to identify patterns from object-oriented systems using software product metrics. The kind of patterns that this research aims at identifying are the previously described GoF design patterns that consist of 23 patterns catalogued in the pattern book [45].

This chapter is organised as follows:

Section 1 explains the reasons why it is necessary to detect pattern instances from existing systems. Then, it is followed by a study on OO development and maintenance model that is fundamentally based on OOA/D and OOP. In Section 3, a pattern re-

covery method is developed and its implications are studied in detail with a specific example. Finally, a summary of the the DPR method is given in the end.

3.1 Introduction

Software design patterns are a way of facilitating design reuse in object-oriented systems by capturing recurring design practices. For several years, people from both industry and academia have discovered many design patterns. Among them, the GoF patterns are the most popular. Also, people are finally realising various usages of patterns, e.g., documenting frameworks [57] and reengineering legacy systems [68, 129]. Furthermore, patterns and pattern languages are used to improve software processes and software organisational structures [129, 35]. To maximise the benefits of using this new abstraction and structuring concept, it is essential to develop a more systematic method to detect patterns. While it is equally important to discover new patterns, for this research the focus is only on detecting GoF patterns.

3.2 OO Software Development and Maintenance Model

In OO methods, problem space and solution space are linked closely through various features such as static ones like *encapsulation* and *inheritance* and more dynamic ones like *message passing* and *object composition* [23]. Four different worlds are assumed

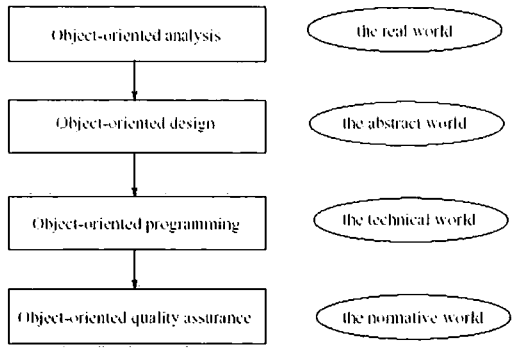

Figure 3.1: Four different worlds represented in OO systems

in object-oriented systems, i.e., *the real world, the abstract world, the technical world* and *the normative world* [86] as shown in Figure 3.1. Each of these are associated with their respective tasks. Thus it is right to say that developing OO systems is essentially an evolutionary process. All these worlds are potentially fruitful sources of discovering patterns. In most existing work, however, the emphasis has been on examining design and programming artifacts of the abstract and technical worlds, and this is where this research work has focussed.

Although many variants of software life cycle exist, normally software is developed starting from defining a problem, via finding a solution for the problem, finally, to implementing it. Thus recovering the design information needs a reverse process of the software development steps.

Along with the system-wide software life cycle, classes themselves have their own life cycle. The Fractal Model proposed by Foote explains this well [41]. His model differentiates three distinct stages that a typical class goes through.

The first stage is called a *Prototype*, or *Initial Design Phase*. This is a quick first pass that may be quite loosely structured, and makes use of expedient, inheritance-based code borrowing. During this stage, the designer should concentrate on the problem at hand, and reuse is his secondary concern.

If an object proves successful, then it enters an *Expansionary*, or *Exploratory Design Phase*. Foote argues that there is a distinctly Darwinian quality about this. Because the object has demonstrated utility, users of the object attempt to reuse it in ways that differ from its original purpose to varying degrees. In conventional systems, such reuse might be undertaken by scavenging copies of the original component, or by introducing flags and conditionals into the original code. These kinds of activities result in destroying the system's structure and behaviour.

Object-oriented systems can retain the integrity of the original code by placing new code in subclasses. As a result, broad, shallow white-box class hierarchies are developed. The subclasses added during the exploratory phase preserve the integrity and identity of the requirements that inspired them, but are not yet truly general.

During the *Consolidation*, or *Design Generalisation Phase*, experience accrued during successive reapplications of an object is used to increase its generality and structural integrity. During this phase, the programmer reorganises the class hierarchy, and abstract classes that reflect the structural regularities in the system and the problem domain emerge. The informal, inheritance-based, white-box relationships that may

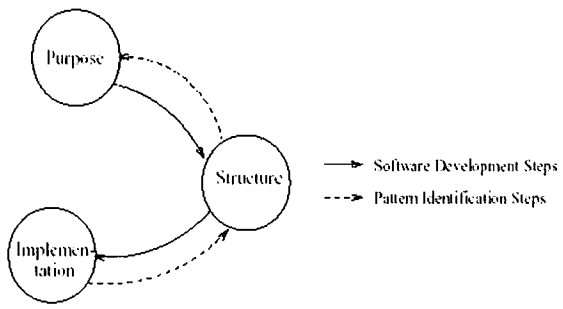

Figure 3.2: Software development and pattern investigation steps

be present in the system can be recast using black-box components. Consolidation is undertaken in an opportunistic fashion, when the insight to justify refactoring has been developed.

These three phases of evolving classes can be useful for designing frameworks as they attempt to capture the reuse potentials of classes and objects during software development.

Shull et al [119] identified three major parts comprising any design patterns on the basis of the descriptions used in GOF's pattern catalogue. They are "purpose", "structure" and "implementation". Figure 3.2 shows the opposite directions that software development and pattern identification steps take, respectively.

As indicated in Figure 3.3, design pattern recovery can bring greater benefits than the normal reverse engineering process as the former captures more fragments of design information than the latter one. Further, the design information captured in design pattern recovery process offers greater grains and is in a more formal fashion than the

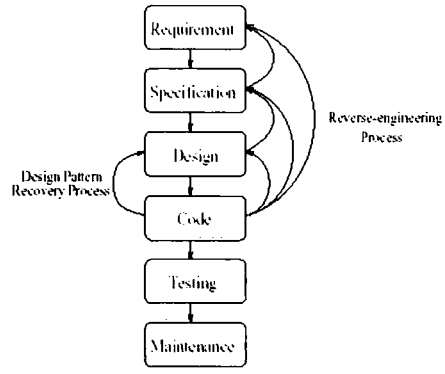

Figure 3.3: The two types of reverse engineering processes

software knowledge obtained through more traditional reverse engineering processes.

There are many ways of representing OO systems either formally or informally. One of the most helpful representations for this research purpose is the one that views an OO system as "a collection of classes interacting with each other in the form of design patterns". Ideally, an object-oriented system can be wholly represented with design patterns. However, usually this is not the case. It is more common that some portions of an OO system are implemented in a less organised manner rather than following the pattern-based approach. This is why it is necessary to restructure software in order to reveal design rationales more clearly. Further, the improvements of programming languages have resulted in the situations where certain patterns can be directly realised just by following the language rules. For example, classes and objects might be called patterns in some traditional language environments like COBOL or C. This indicates the need for more explicit language support with regards to the easy application of design patterns [25, 50, 26, 36].

3.3 The Design Pattern Recovery (DPR) Method

Design pattern matching is essentially a *bipartite graph matching problem* [95]. That is, in the case of pattern detection, a mapping needs to be established between a group of classes/objects that comprise a pattern to another group in a system. Below a more detailed explanation of this situation is given.

To begin with, any simple graph, G can be identified as consisting in a finite set of nodes, say V, generally referred to as vertices and a set, say E, which contains as elements subsets of V, each subset consisting of a pair of vertices which may join together and refer to as an edge [109]. Thus a graph, G is essentially the pair of sets V and E that is to say: $G = (V, E)$.

Consider a simple case where a graph G consists of four vertices, $V = \{v_1, v_2, v_3, v_4\}$. If it is assumed for simplicity that this graph does not contain repeated edges, i.e., is not a multigraph, the possible edges of a graph constructed from the Cartesian product of the vertices of the graph are $E \subseteq V \times V$. The above subset does allow of directional edges, such as $E = \{[v_1, v_2], [v_2, v_1]\}$, and such graphs are referred as *digraphs*. For instance, given

$V = \{v_1, v_2, v_3, v_4\}$, and $E = \{[v_1, v_2], [v_2, v_3], [v_3, v_4], [v_4, v_1], [v_1, v_3], [v_4, v_2]\}$,

the graph $G = (\{v_1, v_2, v_3, v_4\}, \{[v_1, v_2], [v_2, v_3], [v_3, v_4], [v_4, v_1], [v_1, v_3], [v_4, v_2]\})$ is produced. Diagrammatically, this may be represented as Figure 3.4.

A bipartite graph possesses the quality that the set of vertices V, of the graph

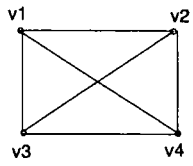

Figure 3.4: Representation of a graph $G = (V, E)$

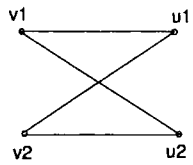

Figure 3.5: Representation of a bipartite graph $G = (V, U, E)$

$G = (V, E)$ may be considered as being divided into two distinct sets, say V and U. Any individual edge, e such that $e \in E$ has one of its pairs from the set V and another from the set U such that a mapping from the elements of V to those of U is obtained, $V \leftrightarrow U$. E may now be considered as a subset of the Cartesian product, $V \times U$, that is $E \subseteq V \times U$. Suppose there is a bipartite graph, G, such that $G = (V, U, E)$ where $V = \{v_1, v_2\}$ and $U = \{u_1, u_2\}$. An arbitrary value may be assigned to E in a similar way to above to obtain all mapping permutations from V to U. This should give a set E with the cardinality of $V \times U$, that is $|E| = |(V \times U)|$.

$E = \{[v_1, u_1], [v_2, u_2], [v_2, u_1], [v_2, u_2]\}$ and

$G = (\{v_1, v_2\}, \{u_1, u_2\}, \{[v_1, u_1], [v_2, u_2], [v_2, u_1], [v_2, u_2]\})$ are obtained. Diagrammatically, this may be represented as Figure 3.5.

In the case of design pattern matching, the set V will be the collection of classes comprising each pattern while the second set, U is the whole collection of classes in a

system where users are interested in finding patterns. Obviously, this process is computationally complex and expensive in general. The worst case complexity is O (n!) as every possible permutation may have to be explored [77]. In addition, a class does not necessarily participate in only one pattern. As it is known, the same class can be used to realise other patterns. Clearly, this phenomenon makes it more difficult to map the classes of a pattern to another group of classes in a system. One solution to solve this problem is finding *characteristic* or *functionally dominant* classes or objects among the ones that comprise patterns. This kind of classes are called *c-classes* hereafter. Users of this method can simply select a class having a high metric as a characteristic one, or expressed in a better way, choose a characteristic class by analysing it semantically. For example, the Composite pattern consists of several classes as shown in Figure 3.6. Among them, the class named "Composite" is really essential for the pattern to carry out its tasks and achieve its goals so that this class may be more characteristic than others with regards to achieving its own goal of the pattern usage. Therefore using this kind of information is semantically more viable.

For this work there are three steps involved in the DPR method. First, the GQM approach is used to model the measurement plan for detecting pattern existence. Then, c-classes of each GoF pattern are identified by inspecting their description appearing in the pattern book in the format of the GoF pattern template. Finally, on the basis of the measurement plan established and the c-classes identified, a pattern matching algorithm is developed by applying some statistical analyses. The matching algorithm

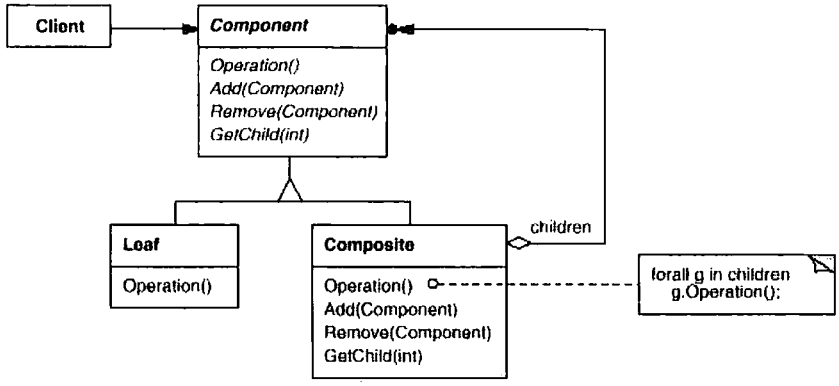

Figure 3.6: The structure of the Composite pattern

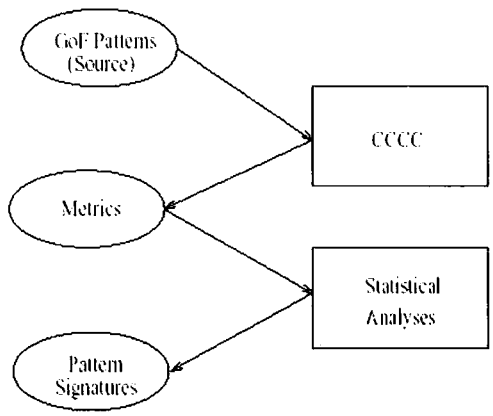

Figure 3.7: The process to extract pattern signatures

utilises *p-value*, *s-value* and a weighting scheme for different kinds of metrics based on heuristics and experience[1]. In addition, each set of p-values comprises their respective *signatures* for identifying themselves uniquely. Figure 3.7 shows these steps involved in extracting signatures for pattern matching.

[1]p-value and s-value are named such because they are associated with *percentile* and *similarity*, respectively.

Below the above steps are explored in greater details.

3.3.1 Applying the Goal/Question/Metric (GQM) Method

In OO systems design information can be recovered more easily than in the ones programmed in procedural languages. This is because in the former, semantic information and syntactic information are more closely associated than in the latter. It is possible that by measuring and analysing the syntactic characteristics of software, the semantic information embedded in those syntactic program structures can be obtained. However, using only procedural metrics and structural metrics is insufficient for this kind of task as OO programs do not follow the traditional way of software building. OO metrics should be collected along with the other two groups of metrics to recover patterns properly.

Establishing a proper measurement plan is important in order to achieve the goal that is aimed at, and to measure what is intended. The GQM method is one of the most popular ways to plan a measurement scheme. A detailed explanation of it has already been given in Chapter 2, and it is used here for the purpose to recover patterns from OO systems and evaluate the PBR method itself. In this research, the GQM plan can be simply established as follows:

Goal 1: Recover design patterns.

Question 1: What are the main constituents of an OO system? – classes, objects as

their dynamic instances, and their various relationships like aggregation, association, and generalisation.

Metrics 1: procedural metrics – to measure intra-module properties.

Metrics 2: structural metrics – to measure inter-module properties.

Metrics 3: object-oriented metrics – to measure OO properties.

Question 2: What are the building blocks that design patterns are implemented with? – classes, objects as their dynamic instances, and their various relationships like aggregation, association and generalisation.

Metrics 1: procedural metrics – to measure intra-module properties.

Metrics 2: structural metrics – to measure inter-module properties.

Metrics 3: object-oriented metrics – to measure OO properties.

Goal 2: Determine the effectiveness and correctness of the DPR method[2].

Question 3: How accurate is the method at picking out design patterns?

Metrics 4: positive true cases and negative false cases

[2]As in the most typical pattern matching examples, there are four different occasions when dealing with recovered pattern candidates. They are *positive true*, *positive false*, *negative true*, and *negative false*. The first two are about deciding the trueness of identified pattern candidates, whereas the rest two handle the trueness of pattern instances that were not recovered as pattern candidates.

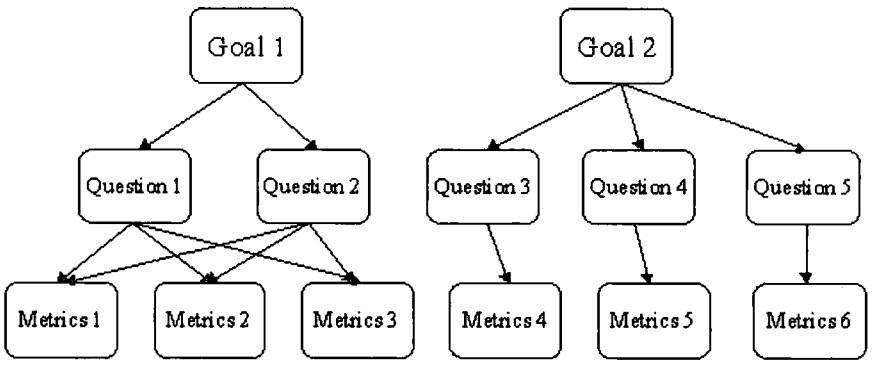

Figure 3.8: An instantiation of the GQM method for the DPR method

Question 4: Does the method pick out patterns that are not there?

Metrics 5: positive false cases

Question 5: Does the method fail to find patterns that are there?

Metrics 6: negative true cases

As indicated above, in the case of this research the first goal is quite clear, i.e., to recover patterns. Then, to address the goal two questions can be asked. One is about an OO system, and another about a pattern as patterns are identified from OO systems. The same set of metrics are assigned to these two GQM questions.

Software systems developed in languages like C++ and Java mainly consist of classes, their dynamic instances, i.e., objects, associations, generalisations and interactions between these. Also if a thorough investigation is made on them, it can be observed that they are based on more traditional language features such as variables, operators,

conditional branches, functions, and procedures. Thus, if people want to know the structure and behaviour of an OO system, they should consider the three kinds of metrics that were mentioned above. The following are the metrics that have been adopted in this pattern investigation:

1. Procedural metrics

   - Lines of code (LOC): This is one of the oldest measures that simply counts the number of non-blank, non-comment lines of source code. Many people have argued against the usefulness of this fairly simple metric because of its limitations in terms of predicting many different system characteristics. However, it is still highly likely that code portions containing larger and more important information like the one included in a *c-class* require more LOC than others playing minor roles in the pattern instances.
   - McCabe's cyclomatic complexity (MVG): Cyclomatic complexity is defined for each module to be $e - n + 2$, where e and n are the number of edges and nodes in the control flow graph, respectively. Cyclomatic complexity is also known as $v(G)$, where v refers to the cyclomatic number in graph theory and G indicates that the complexity is a function of the graph [81, 143]. This was developed to overcome the weakness of LOC. As with LOC, MVG is a useful metric for locating which parts of code are more complex, and therefore likely to contain relevant information.

- Lines of comments (COM): This is the counting of number of lines of comments, and can be used to guess the quantity of design information contained within a specific portion of code. Although at the worst case programmers do not bother to put comments into their code irrespective of the importance and complexity of the code. It is a common practice that most well-trained professionals put a reasonable amount of comments proportional to the importance and complexity of code they are writing.

2. OO metrics

These four different metrics were developed by Chidamber and Kemerer at MIT Slone Business School in the early 1990s [31]. It is interesting to know that the original usage of these OO metrics was to assist managerial decision making in IT sectors. They are as follows:

- Weighted methods per class (WMC): This measures the sum of a weighting function over the functions of a module. Two different weighting functions are applied: WMC1 uses the nominal weight of 1 for each function, and hence measures the number of functions, while WMCv uses a weighting function which is 1 for functions accessible to other modules, 0 for private functions. The main weakness of this metric is that it does not address the different types of methods or operations[3]. An operation denotes a service

[3]In UML terms, this kind of type is called *stereotype* that is an extension of the vocabulary of the UML, which allows users to create new kinds of building blocks that are derived from existing ones but

that a class offers to its clients. Booch classifies the five most common kinds of operations as follows [23]:

- Modifier: An operation that alters the state of an object.
- Selector: An operation that accesses the state of an object, but does not alter the state.
- Iterator: An operation that permits all parts of an object to be accessed in some well-defined order.
- Constructor: An operation that creates an object and/or initialises its state.
- Destructor: An operation that frees the state of an object and/or destroys the object itself.

Lippman suggests a slightly different categorisation: manager functions, implementor functions, helping functions (all kinds of modifiers), and access functions (equivalent to selectors) [75].

WMC is not concerned about these classifications but it simply differentiates between private operations and non-private ones. The rationale behind the inclusion of this metric despite its drawback is that operations are one of few ways through which different parts of an OO system can communicate. By including this metric it is possible to predict the characteristics of the class where these operations belong.

---

are specific to their problem [24].

- Depth of inheritance tree (DIT): This is the measure of the length of the longest path of inheritance ending at the current module. The deeper the inheritance tree for a module, the harder it may be to predict its behaviour. On the other hand, increasing depth gives the potential of greater reuse by the current module of behaviour defined for ancestor classes. This metric is concerned about the vertical hierarchy of generalisation of a system. As generalisation is one of major building blocks of a pattern, this is an especially useful metric.

- Number of children (NOC): This counts the number of modules which inherit directly from the current module. Moderate values of this measure indicate scope for reuse, however high values may indicate an inappropriate abstraction in the design. While DIT is for measuring the vertical hierarchy of a generalisation relationship, NOC measures its horizontal aspect. As observed in some structural patterns like *Facade*, NOC can be a useful measure for detecting the essentiality of a pattern.

- Coupling between objects (CBO): This is the measure of the number of other modules which are coupled to the current module either as a client or a supplier. Excessive coupling indicates weakness of module encapsulation and may inhibit reuse. This metric has been used for detecting an undesirable property of a system or a component as tightly coupled classes make it difficult to maintain them. However, in this research quality as-

pects of a system are not dealt with but only the detection of patterns is addressed. CBO can be a good candidate metric for figuring out collaborations between classes because patterns are basically a group of collaborating classes/objects achieving their own specific goals.

3. Structural metrics

There exist three variants of each of the structural metrics: a count restricted to the part of the interface which is externally visible (FIv, FOv and IF4v), a count which only includes relationships which imply the client module needs to be recompiled if the supplier's implementation changes (FIc, FOc and IF4c), and an inclusive count (FIi, FOi and IF4i), where FI, FO and IF4 are respectively defined as follows [34, 55]:

- Fan-in (FI): This measures the number of other modules which pass information into the current module.
- Fan-out (FO): This is obtained by counting the number of other modules into which the current module passes information.
- Information flow measure (IF4): This is a composite measure of structural complexity, calculated as the square of the product of the fan-in and fan-out of a single module.

Although FI, FO and IF4 are grouped as structural metrics, they actually capture static snapshots of dynamic features going on between modules in a system. Dif-

ferent patterns cause different degrees of actions thus these metrics are included for pattern detection.

3.3.2 Characteristic Classes of Patterns

Characteristic class of a pattern is defined as the most important and thus functionally dominant class among the many participants of the pattern. The concept of c-class was originally proposed by Spanoudakis and it was used to measure the significance of an inconsistency in object-oriented software development [125].

For example, consider the Abstract Factory pattern. This pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes [45]. Figure 3.9 shows how each constituent of the pattern collaborates to achieve the intended aims of the pattern. Obviously, the class, *AbstractFactory* seems to be the most important one among the pattern participants. Thus it is not a surprise that the pattern is named such.

Table 3.1 indicates the characteristic classes of each pattern identified from the design pattern examples published in the design pattern book [45]. They were obtained by inspecting the sections of each pattern carefully. This approach has an advantage over simply choosing classes having maximum or minimum metric values because it makes use of a certain degree of semantic information.

Not all patterns are suitable for detection in the above way though. Especially, in the

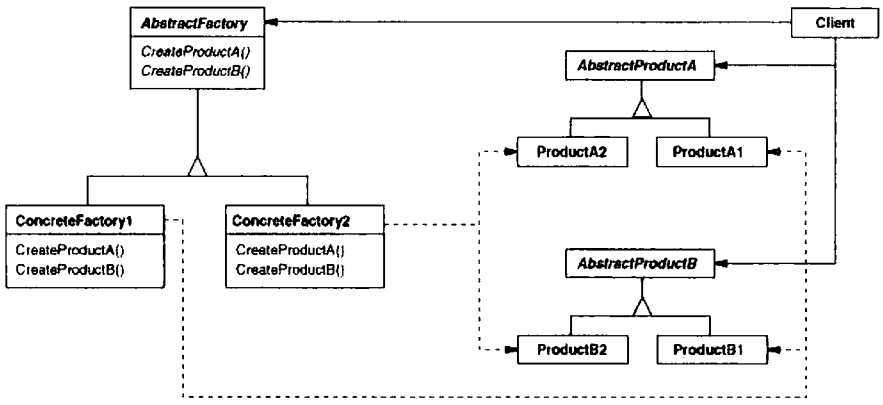

Figure 3.9: The structure of the Abstract Factory pattern (from Gamma et al. [45])

Pattern	C-Class	Pattern	C-Class
Abstract Factory	MazeFactory	Iterator	Iterator
Adapter	TextShape	Mediator	DialogDirector
Bridge	WindowImp	Memento	Memento
Builder	MazeBuilder	Observer	Observer
Chain of Responsibility	HelpHandler	Prototype	MazePrototypeFactory
Command	Command	Proxy	ImageProxy
Composite	CompositeEquipment	Singleton	Singleton
Decorator	Decorator	State	TCPState
Facade	Compiler	Strategy	Compositor
Factory Method	MazeGame	Template Method	View
Flyweight	Glyph	Visitor	Visitor
Interpreter	BooleanExp		

Table 3.1: c-classes of each pattern example from the pattern book [45]

case of the Singleton pattern, it would certainly be better to search for a static instance operation and a static member that holds the one and only instance.

3.3.3 The Pattern Matching Algorithm Using P-Values and S-Values

In the previous sections the GQM plan for this investigation was developed and c-classes of each pattern were identified. Now, it is necessary to develop an algorithm to map each design pattern to their corresponding metrics patterns that are unique to each other. First, using CCCC[4], the GoF patterns examples appearing in the pattern book [45] are processed to obtain various software metrics information. Then some statistical analysis techniques are applied to them. Obviously, this mapping scheme for allotting real data to formal data, i.e., metrics, is comparative. As the size of classes varies greatly depending on each program, it is meaningless to allot absolute metrics to each pattern. Among many plausible statistical analysis methods, *rank* and *percentile* analyses were selected. These analyses produce a table that contains the ordinal and percentage rank of each value in a data set. The relative standing of the values in a data set can be analysed. For example, Tables 3.2 to 3.4 show how p-values of each CBO metric value can be obtained. The data appearing in column 1 to 5 represent the names of the classes, their CBO metric values, ranks, percentile values, and p-values, respectively. The p-value of a metric value is determined by its rank and percentile

[4]CCCC (C and C++ Code Counter) is a metrics producer for the C language and the C++ language. It was developed by Tim Littlefair in Australia in 1997.

value. The Data Analysis facility of Microsoft Excel 97 is used to generate p-values semi-automatically[5]. They are called p-values as they originate from percentile values and the range of the values are between 0 and 1 inclusive. In this transformation, the maximum value corresponds to 1 whereas 0 is for the minimum value. P-values are not absolute but relative to the metric values of other classes in a system. This approach of using ranks and percentile values is better than using normal distribution curve generated by computing average and standard deviation regarding the breadth of the values obtained. It was observed that some metrics do not show the normal distribution. One of the reasons for this phenomenon is that unlike class libraries or OO frameworks, application programs are not carefully planned and thought out when designed in the first place. For example, class libraries like the Java class libraries have inheritance hierarchies that are both much broader and deeper than normal programs, resulting in a high variance of DIT and NOC metrics.

More formally, p-value can be defined as follows.

Definition 1: The p-value of a class c_i is defined as:

$$p_m(c_i) = \frac{Percentile \circ Rank(Metric_m(c_i))}{100}$$

where

[5] *Excel 97* is a registered trademark of *Microsoft Corporation.*

Module Name	CBO	*Rank*	*Percentile*	P-Value	Module Name	CBO	*Rank*	*Percentile*	P-Value
ASCII7Stream	1	131	5.50%	0.06	Coord	1	131	5.50%	0.06
AStrategy	1	131	5.50%	0.06	CountingMazeBuilder	1	131	5.50%	0.06
AbstractList	0	171	.00%	0.00	Creator	4	35	67.00%	0.67
AnalogClock	4	35	67.00%	0.67	Currency	1	131	5.50%	0.06
AndExp	2	86	27.90%	0.28	Decorator	3	61	53.00%	0.53
Application	4	35	67.00%	0.67	DerivedClass	1	131	5.50%	0.06
ArrayCompositor	1	131	5.50%	0.06	Dialog	3	61	53.00%	0.53
BTree	1	131	5.50%	0.06	DialogDirector	6	9	89.30%	0.89
BaseClassSubject	1	131	5.50%	0.06	DigitalClock	4	35	67.00%	0.67
Body	1	131	5.50%	0.06	Document	2	86	27.90%	0.28
BombedMazeFactory	1	131	5.50%	0.06	Door	5	21	81.50%	0.82
BombedMazeGame	1	131	5.50%	0.06	DoorNeedingSpell	3	61	53.00%	0.53
BombedWall	2	86	27.90%	0.28	Element	4	35	67.00%	0.67
BooleanExp	6	9	89.30%	0.89	ElementA	3	61	53.00%	0.53
BorderDecorator	2	86	27.90%	0.28	ElementB	3	61	53.00%	0.53
Bus	4	35	67.00%	0.67	Employee	0	171	.00%	0.00
Button	5	21	81.50%	0.82	EnchantedMazeBuilder	3	61	53.00%	0.53
BytecodeStream	3	61	53.00%	0.53	EnchantedMazeFactory	2	86	27.90%	0.28
Cabinet	1	131	5.50%	0.06	EnchantedMazeGame	2	86	27.90%	0.28
Card	4	35	67.00%	0.67	EnchantedRoom	2	86	27.90%	0.28
Character	4	35	67.00%	0.67	EntryField	4	35	67.00%	0.67
Chassis	6	9	89.30%	0.89	Equipment	5	21	81.50%	0.82
ClockTimer	3	61	53.00%	0.53	EquipmentVisitor	10	4	98.30%	0.98
CodeGenerator	6	9	89.30%	0.89	Event	3	61	53.00%	0.53
Collection	2	86	27.90%	0.28	ExpressionNode	5	21	81.50%	0.82
Column	0	171	.00%	0.00	ExtendedHandler	2	86	27.90%	0.28
Command	3	61	53.00%	0.53	FileStream	1	131	5.50%	0.06
Compiler	2	86	27.90%	0.28	FilteringListTraverser	3	61	53.00%	0.53
Component	3	61	53.00%	0.53	FloppyDisk	5	21	81.50%	0.82
Composite	1	131	5.50%	0.06	Font	2	86	27.90%	0.28
CompositeElement	3	61	53.00%	0.53	FontDialogDirector	5	21	81.50%	0.82
CompositeEquipment	6	9	89.30%	0.89	Glyph	4	35	67.00%	0.67
Composition	2	86	27.90%	0.28	GlyphContext	4	35	67.00%	0.67
Compositor	4	35	67.00%	0.67	GlyphFactory	1	131	5.50%	0.06
CompressingStream	2	86	27.90%	0.28	Graphic	9	5	97.20%	0.97
Constant	2	86	27.90%	0.28	Handle	1	131	5.50%	0.06
ConstraintSolver	2	86	27.90%	0.28	Handler	4	35	67.00%	0.67
ConstraintSolverMemento	2	86	27.90%	0.28	HelpHandler	4	35	67.00%	0.67
Context	9	5	97.20%	0.97	HelpRequest	2	86	27.90%	0.28

Table 3.2: CBO metric values of each class in the GoF patterns examples 1/3

Module Name	CBO	*Rank*	*Percentile*	P-Value	Module Name	CBO	*Rank*	*Percentile*	P-Value
Image	7	8	96.00%	0.96	Parser	2	86	27.90%	0.28
ImageProxy	6	9	89.30%	0.89	PasteCommand	2	86	27.90%	0.28
ImagePtr	1	131	5.50%	0.06	Point	6	9	89.30%	0.89
Inventory	2	86	27.90%	0.28	PricingVisitor	6	9	89.30%	0.89
InventoryVisitor	6	9	89.30%	0.89	PrintNEmployees	2	86	27.90%	0.28
Item	3	61	53.00%	0.53	PrintRequest	2	86	27.90%	0.28
ItemType	0	171	.00%	0.00	Product	4	35	67.00%	0.67
IterationState	1	131	5.50%	0.06	ProductId	2	86	27.90%	0.28
Iterator	3	61	53.00%	0.53	ProgramNode	3	61	53.00%	0.53
IteratorPtr	1	131	5.50%	0.06	ProgramNodeBuilder	2	86	27.90%	0.28
Leaf	1	131	5.50%	0.06	RISCCodeGenerator	4	35	67.00%	0.67
List	*14*	1	99.40%	0.99	Request	4	35	67.00%	0.67
ListBox	5	21	81.50%	0.82	ReverseListIterator	2	86	27.90%	0.28
ListIterator	5	21	81.50%	0.82	Room	*14*	1	99.40%	0.99
ListTraverser	4	35	67.00%	0.67	RoomWithABomb	2	86	27.90%	0.28
Manipulator	1	131	5.50%	0.06	Row	0	171	.00%	0.00
MapSite	3	61	53.00%	0.53	Scanner	2	86	27.90%	0.28
Maze	4	35	67.00%	0.67	ScrollDecorator	2	86	27.90%	0.28
MazeBuilder	4	35	67.00%	0.67	Shape	2	86	27.90%	0.28
MazeFactory	5	21	81.50%	0.82	SimpleCommand	1	131	5.50%	0.06
MazeGame	5	21	81.50%	0.82	SimpleCompositor	1	131	5.50%	0.06
MazePrototypeFactory	5	21	81.50%	0.82	Singleton	2	86	27.90%	0.28
Memento	2	86	27.90%	0.28	SkipList	1	131	5.50%	0.06
MouseEvent	4	35	67.00%	0.67	SkipListIterator	2	86	27.90%	0.28
MoveCommand	3	61	53.00%	0.53	Spell	2	86	27.90%	0.28
MyCreator	2	86	27.90%	0.28	StandardCreator	1	131	5.50%	0.06
MyProduct	1	131	5.50%	0.06	StandardMazeBuilder	3	61	53.00%	0.53
MySingleton	1	131	5.50%	0.06	State	2	86	27.90%	0.28
MyStrategy	0	171	.00%	0.00	StatementNode	2	86	27.90%	0.28
MySubject	1	131	5.50%	0.06	Stream	4	35	67.00%	0.67
MyType	1	131	5.50%	0.06	StreamDecorator	2	86	27.90%	0.28
MyView	1	131	5.50%	0.06	Subject	6	9	89.30%	0.89
NameSingletonPair	0	171	.00%	0.00	TCPClosed	2	86	27.90%	0.28
NotExp	2	86	27.90%	0.28	TCPConnection	6	9	89.30%	0.89
Observer	4	35	67.00%	0.67	TCPEstablished	3	61	53.00%	0.53
OpenCommand	2	86	27.90%	0.28	TCPListen	2	86	27.90%	0.28
OrExp	2	86	27.90%	0.28	TCPOctetStream	3	61	53.00%	0.53
Originator	2	86	27.90%	0.28	TCPState	6	9	89.30%	0.89
ParentClass	1	131	5.50%	0.06	TeXCompositor	1	131	5.50%	0.06

Table 3.3: CBO metric values of each class in the GoF patterns examples 2/3

Module Name	CBO	*Rank*	*Percentile*	P-Value	Module Name	CBO	*Rank*	*Percentile*	P-Value
Text	1	131	5.50%	0.06	Visitor	8	7	96.60%	0.97
TextDocument	1	131	5.50%	0.06	VisualComponent	5	21	81.50%	0.82
TextManipulator	2	86	27.90%	0.28	Wall	3	61	53.00%	0.53
TextRange	1	131	5.50%	0.06	Widget	12	3	98.80%	0.99
TextShape	4	35	67.00%	0.67	Window	4	35	67.00%	0.67
TextView	3	61	53.00%	0.53	WindowImp	0	171	.00%	0.00
TheirProduct	1	131	5.50%	0.06	YourProduct	1	131	5.50%	0.06
Token	0	171	.00%	0.00	YourType	1	131	5.50%	0.06
Topic	5	21	81.50%	0.82	anonymous	0	171	.00%	0.00
TwistyTurnyPassage	1	131	5.50%	0.06	bool	4	35	67.00%	0.67
VariableExp	3	61	53.00%	0.53	istream	5	21	81.50%	0.82
View	2	86	27.90%	0.28	ostream	3	61	53.00%	0.53

Table 3.4: CBO metric values of each class in the GoF patterns examples 3/3

- $m \in M$, and M is the set of the 17 kinds of metrics:

$$M = \{LOC, MVG, COM, WMC1, WMCv, DIT, NOC, CBO, FOv, FOc, FOi, FIv, FIc, FIi, IF4v, IF4c, IF4i\}$$

- $Rank(x)$ produces the rank of the value x in a data set.

- $Percentile(x)$ assigns the 100th percentile to the value x if x is the maximum value and 0th percentile to it if it is the minimum value in a data set. Intermediate values have percentiles in steps of $1/(n-1)$, where n is the number of the values in a data set.

- $Metric_m(c_i)$ is the m metric of the class c_i.

Using the above formulae and steps, metrics signatures of each pattern were obtained. They consist of p-values. Table 3.5 shows them all.

Pattern	C-Class	LOC	MVG	COM	WMC1	WMCv	DIT	NOC	CBO	Fov	Foc	Foi	Fiv	Fic	Fii	IF4v	IF4c	IF4i
Abstract Factory	MazeFactory	0.99	1.00	0.98	0.83	0.73	0.00	0.93	0.82	0.88	0.91	0.87	0.31	0.00	0.28	0.89	0.00	0.86
Adapter	TextShape	0.97	0.89	0.97	0.70	0.73	0.90	0.00	0.67	0.52	0.00	0.46	0.85	0.91	0.80	0.84	0.00	0.78
Bridge	WindowImp	0.05	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Builder	MazeBuilder	0.79	0.82	0.63	0.70	0.73	0.00	0.93	0.67	0.88	0.91	0.87	0.00	0.00	0.00	0.00	0.00	0.00
Chain of Responsibility	HelpHandler	0.94	0.86	0.84	0.83	0.73	0.00	0.87	0.67	0.77	0.83	0.75	0.31	0.49	0.28	0.84	0.96	0.78
Command	Command	0.37	0.00	0.32	0.46	0.49	0.00	0.93	0.53	0.77	0.91	0.75	0.00	0.00	0.00	0.00	0.00	0.00
Composite	CompositeEquipment	0.87	0.82	0.86	0.97	0.98	0.53	0.93	0.89	0.77	0.91	0.75	0.63	0.91	0.80	0.93	0.99	0.95
Decorator	Decorator	0.68	0.00	0.73	0.46	0.49	0.53	0.87	0.53	0.68	0.83	0.65	0.31	0.49	0.28	0.77	0.96	0.74
Facade	Compiler	0.72	0.00	0.53	0.36	0.37	0.00	0.00	0.28	0.00	0.00	0.00	0.63	0.00	0.58	0.00	0.00	0.00
Factory Method	MazeGame	0.99	0.99	0.99	0.97	0.98	0.00	0.87	0.82	0.68	0.83	0.65	0.85	0.00	0.80	0.93	0.00	0.91
Flyweight	Glyph	0.69	0.00	0.77	0.99	1.00	0.00	0.76	0.67	0.52	0.71	0.46	0.85	0.00	0.80	0.84	0.00	0.78
Interpreter	BooleanExp	0.43	0.00	0.53	0.70	0.73	0.00	0.99	0.89	0.94	0.99	0.94	0.31	0.00	0.28	0.92	0.00	0.89
Iterator	Iterator	0.65	0.00	0.25	0.70	0.63	0.00	0.87	0.53	0.77	0.83	0.75	0.00	0.00	0.00	0.00	0.00	0.00
Mediator	DialogDirector	0.50	0.00	0.53	0.70	0.73	0.00	0.76	0.89	0.94	0.71	0.94	0.31	0.00	0.28	0.92	0.00	0.89
Memento	Memento	0.58	0.00	0.77	0.61	0.20	0.00	0.00	0.28	0.52	0.00	0.46	0.00	0.00	0.28	0.00	0.00	0.68
Observer	Observer	0.37	0.00	0.00	0.46	0.49	0.00	0.87	0.67	0.77	0.83	0.75	0.31	0.00	0.28	0.84	0.00	0.78
Prototype	MazePrototypeFactory	0.98	0.97	0.92	0.70	0.73	0.53	0.00	0.82	0.00	0.00	0.00	0.98	0.49	0.97	0.00	0.00	0.00
Proxy	ImageProxy	0.95	0.89	0.96	0.94	0.94	0.53	0.00	0.89	0.00	0.00	0.00	0.98	0.91	0.99	0.00	0.00	0.00
Singleton	Singleton	0.92	0.89	0.80	0.61	0.63	0.00	0.76	0.28	0.52	0.71	0.46	0.00	0.00	0.28	0.00	0.00	0.68
State	TCPState	0.82	0.00	0.65	0.94	0.94	0.00	0.93	0.89	0.77	0.91	0.87	0.63	0.00	0.58	0.93	0.00	0.93
Strategy	Compositor	0.68	0.00	0.86	0.46	0.49	0.00	0.93	0.67	0.88	0.91	0.87	0.00	0.00	0.00	0.00	0.00	0.00
Template Method	View	0.75	0.00	0.65	0.70	0.73	0.00	0.76	0.28	0.52	0.71	0.46	0.31	0.00	0.28	0.73	0.00	0.68
Visitor	Visitor	0.62	0.00	0.65	0.70	0.73	0.00	0.00	0.97	0.88	0.00	0.87	0.93	0.00	0.92	0.98	0.00	0.98

Table 3.5: GoF patterns signatures based on c-classes and p-values

Metric	Weight	Metric	Weight	Metric	Weight
LOC	1	NOC	2	FIc	1
MVG	2	CBO	2	FIi	1
COM	1	FOv	1	IF4v	1
WMC1	1	FOc	1	IF4c	1
WMCv	1	FOi	1	IF4i	1
DIT	2	FIv	1		

Table 3.6: The weights assigned to each metric

Note that the same set of metrics were allotted to Questions 1 and 2 of Goal 1 in the GQM plan developed in Chapter 2. This implies that the subject system that users are interested in needs to be measured and compared to pattern metrics signatures. The similarity between both sets of p-values is explored. Different weights were assigned to the 17 kinds of metrics according to their importance to pattern composition as shown in Table 3.6. These weights were decided by the importance of the design concepts, experience and heuristics regarding the formation of a pattern. If these weights are applied to the absolute values of the difference between the two pairs of p-values and the sum of these values is divided by the sum of the weights, then the s-values ranging from 0 to 1 inclusive are obtained. They are called s-values as they measure similarity between two classes.

This is expressed in the following definition.

Definition 2: The s-value between two classes c_i and c_j is defined as:

$$s(c_i, c_j) = \frac{\sum_{m \in M} (w_m \cdot |p_m(c_i) - p_m(c_j)|)}{\sum_{m \in M} w_m}$$

where

- w is the weights of each metric $m \in M$.
- $p_m(c_i)$ and $p_m(c_j)$ are as defined in Definition 1.
- The set M is as defined in Definition 1.

For example, consider detecting patterns from a system called **System 1**, and it has a class called **Class A**. Using the steps described earlier, suppose the p-values of each metric of **Class A** were obtained. They are 0.59, 0.00, 0.00, 0.37, 0.19, 0.00, 0.00, 0.26, 0.63, 0.00, 0.57, 0.00, 0.23, 0.28, 0.00, 0.00, and 0.76, respectively. These p-values are compared with the p-values in Table 3.5 according to the formulae in order to get s-values appearing in the last column of Table 3.7. These s-values indicate the likelihood of a class being an active participant, in other words, c-class, of a pattern instance.

Pattern	LOC	MVG	COM	WMC1	WMCv	DIT	NOC	CBO	Fov	Foc	Foi	Fiv	Fic	Fii	IF4v	IF4c	IF4i	S-Value
Abstract Factory	0.40	1.00	0.98	0.46	0.53	0.00	0.93	0.56	0.26	0.91	0.30	0.31	0.23	0.00	0.89	0.00	0.10	0.49
Adapter	0.38	0.89	0.97	0.33	0.53	0.90	0.00	0.41	0.11	0.00	0.11	0.85	0.68	0.52	0.84	0.00	0.02	0.46
Bridge	0.54	0.00	0.00	0.37	0.19	0.00	0.00	0.26	0.63	0.00	0.57	0.00	0.23	0.28	0.00	0.00	0.76	0.19
Builder	0.20	0.82	0.63	0.33	0.53	0.00	0.93	0.41	0.26	0.91	0.30	0.00	0.23	0.28	0.00	0.00	0.76	0.42
Chain of Responsibility	0.35	0.86	0.84	0.46	0.53	0.00	0.87	0.41	0.14	0.83	0.18	0.31	0.26	0.00	0.84	0.96	0.02	0.48
Command	0.22	0.00	0.32	0.09	0.30	0.00	0.93	0.27	0.14	0.91	0.18	0.00	0.23	0.28	0.00	0.00	0.76	0.28
Composite	0.28	0.82	0.86	0.60	0.79	0.53	0.93	0.63	0.14	0.91	0.18	0.63	0.68	0.52	0.93	0.99	0.19	0.64
Decorator	0.09	0.00	0.73	0.09	0.30	0.53	0.87	0.27	0.05	0.83	0.09	0.31	0.26	0.00	0.77	0.96	0.03	0.37
Facade	0.13	0.00	0.53	0.01	0.17	0.00	0.00	0.02	0.63	0.00	0.57	0.63	0.23	0.30	0.00	0.00	0.76	0.19
Factory Method	0.41	0.99	0.99	0.60	0.79	0.00	0.87	0.56	0.05	0.83	0.09	0.85	0.23	0.52	0.93	0.00	0.14	0.54
Flyweight	0.11	0.00	0.77	0.63	0.81	0.00	0.76	0.41	0.11	0.71	0.11	0.85	0.23	0.52	0.84	0.00	0.02	0.38
Interpreter	0.16	0.00	0.53	0.33	0.53	0.00	0.99	0.63	0.31	0.99	0.37	0.31	0.23	0.00	0.92	0.00	0.13	0.38
Iterator	0.06	0.00	0.25	0.33	0.44	0.00	0.87	0.27	0.14	0.83	0.18	0.00	0.23	0.28	0.00	0.00	0.76	0.28
Mediator	0.09	0.00	0.53	0.33	0.53	0.00	0.76	0.63	0.31	0.71	0.37	0.31	0.23	0.00	0.92	0.00	0.13	0.35
Memento	0.01	0.00	0.77	0.25	0.01	0.00	0.00	0.02	0.11	0.00	0.11	0.00	0.23	0.00	0.00	0.00	0.08	0.08
Observer	0.22	0.00	0.00	0.09	0.30	0.00	0.87	0.41	0.14	0.83	0.18	0.31	0.23	0.00	0.84	0.00	0.02	0.27
Prototype	0.39	0.97	0.92	0.33	0.53	0.53	0.00	0.56	0.63	0.00	0.57	0.98	0.26	0.69	0.00	0.00	0.76	0.48
Proxy	0.36	0.89	0.96	0.57	0.75	0.53	0.00	0.63	0.63	0.00	0.57	0.98	0.68	0.71	0.00	0.00	0.76	0.53
Singleton	0.33	0.89	0.80	0.25	0.44	0.00	0.76	0.02	0.11	0.71	0.11	0.00	0.23	0.00	0.00	0.00	0.08	0.30
State	0.23	0.00	0.65	0.57	0.75	0.00	0.93	0.63	0.14	0.91	0.30	0.63	0.23	0.30	0.93	0.00	0.17	0.43
Strategy	0.09	0.00	0.86	0.09	0.30	0.00	0.93	0.41	0.26	0.91	0.30	0.00	0.23	0.28	0.00	0.00	0.76	0.32
Template Method	0.17	0.00	0.65	0.33	0.53	0.00	0.76	0.02	0.11	0.71	0.11	0.31	0.23	0.00	0.73	0.00	0.08	0.26
Visitor	0.03	0.00	0.65	0.33	0.53	0.00	0.00	0.71	0.26	0.00	0.30	0.93	0.23	0.64	0.98	0.00	0.22	0.31

Table 3.7: s-values of **Class A**

The values in the metrics columns are the absolute values of the difference between two pairs of p-values, while the last column contains their s-values. According to the results appearing in the table, it can be assumed that **Class A** is highly likely to be a participant of a Memento pattern instance as its s-value is only 0.08, whereas it is unlikely that the class is included in a Composite pattern instance having the s-value of 0.64.

Also an investigation was carried out to see how similar each pattern is to others by calculating their s-values. Table 3.8 shows their s-values. According to the s-values in the table, each pattern has a reasonably high degree of distinctiveness or uniqueness in their metrics characteristics.

Because there are 23 patterns, the number of occasions are equal to the number of 2-combinations of a set with 23 distinct elements, i.e.,

$$C\,(23,2) = \frac{23!}{2!(23-2)!} = 253$$

146 s-values are greater than equal to 0.30, thus the percentage is 57.71%. This rate shows the reasonably high degree of *uniqueness* or *distinctiveness* of the pattern signatures of each metric. In Table 3.8, abbreviations were used to display all mappings properly. They are indicated in Table 3.9.

The metrics similarity is commutative but not transitive. This means that when two classes have high similarity and one of them has also high similarity with the third one, that this does not imply that the first one has high similarity with the third one.

Pattern	AB	AD	BR	BU	CH	CM	CP	DE	FA	FT	FY	IN	IT	MD	MM	OB	PR	PX	SI	ST	SY	TE	VI
AB	0.00	0.39	0.66	0.18	0.13	0.33	0.25	0.34	0.55	0.10	0.27	0.18	0.32	0.19	0.45	0.24	0.45	0.49	0.22	0.18	0.27	0.26	0.35
AD		0.00	0.64	0.49	0.37	0.62	0.33	0.44	0.46	0.33	0.37	0.53	0.58	0.49	0.42	0.51	0.22	0.22	0.40	0.47	0.58	0.44	0.34
BR			0.00	0.49	0.67	0.33	0.84	0.57	0.18	0.71	0.53	0.54	0.35	0.51	0.22	0.42	0.49	0.54	0.45	0.60	0.39	0.41	0.48
BU				0.00	0.22	0.16	0.37	0.37	0.43	0.28	0.33	0.25	0.14	0.26	0.37	0.28	0.43	0.48	0.18	0.27	0.12	0.27	0.43
CH					0.00	0.36	0.18	0.21	0.55	0.19	0.28	0.26	0.32	0.26	0.45	0.25	0.47	0.50	0.23	0.26	0.32	0.26	0.42
CM						0.00	0.51	0.27	0.32	0.42	0.31	0.21	0.04	0.23	0.28	0.14	0.57	0.61	0.27	0.27	0.06	0.23	0.42
CP							0.00	0.27	0.66	0.21	0.33	0.35	0.49	0.36	0.62	0.41	0.44	0.39	0.41	0.25	0.47	0.43	0.44
DE								0.00	0.45	0.38	0.27	0.26	0.27	0.25	0.36	0.20	0.53	0.58	0.34	0.28	0.27	0.21	0.34
FA									0.00	0.53	0.36	0.45	0.32	0.41	0.16	0.39	0.32	0.36	0.36	0.42	0.34	0.29	0.32
FT										0.00	0.18	0.27	0.38	0.27	0.49	0.30	0.39	0.41	0.26	0.18	0.37	0.30	0.33
FY											0.00	0.21	0.27	0.17	0.31	0.19	0.47	0.47	0.28	0.13	0.27	0.13	0.23
IN												0.00	0.21	0.04	0.36	0.12	0.60	0.63	0.34	0.10	0.20	0.18	0.24
IT													0.00	0.21	0.27	0.16	0.53	0.57	0.24	0.25	0.08	0.19	0.38
MD														0.00	0.32	0.12	0.56	0.59	0.30	0.11	0.21	0.14	0.20
MM															0.00	0.31	0.43	0.47	0.23	0.39	0.28	0.20	0.27
OB																0.00	0.61	0.65	0.30	0.17	0.18	0.16	0.30
PR																	0.00	0.06	0.41	0.55	0.53	0.53	0.39
PX																		0.00	0.46	0.53	0.57	0.57	0.42
SI																			0.00	0.34	0.26	0.16	0.46
ST																				0.00	0.23	0.19	0.21
SY																					0.00	0.23	0.38
TE																						0.00	0.30
VI																							0.00

Table 3.8: s-values for showing similarity between patterns

Pattern	Abbreviation	Pattern	Abbreviation
Abstract Factory	AB	Iterator	IT
Adapter	AD	Mediator	MD
Bridge	BR	Memento	MM
Builder	BU	Observer	OB
Chain of Responsibility	CH	Prototype	PR
Command	CM	Proxy	PX
Composite	CP	Singleton	SI
Decorator	DE	State	ST
Facade	FA	Strategy	SY
Factory Method	FT	Template Method	TE
Flyweight	FY	Visitor	VI
Interpreter	IN		

Table 3.9: Patterns and their abbreviations

Another investigation has been performed to see whether patterns belonging to the same category, i.e., one of the three categories, creational, structural, and behavioural, have a strong correlation between them. This has been measured by computing their s-values. According to the results shown in Tables 3.10, 3.11, and 3.12, the percentages of s-values between patterns belonging to a same category that are less than 0.30 were 60.00%, 19.05%, and 74.55% for creational, structural, and behavioural categories, respectively. Thus, it can be said that in the case of creational and behavioural patterns there exists a high correlation between patterns belonging to a same category regarding their metrics characteristics.

Low s-values between patterns increase the chance of negative true and positive false occurrences. This results from the fact that this metrics-based pattern detection method is not perfect. Therefore at the final stage of this method, a filtering or checking process needs to be performed. As with most software engineering problem solving

Pattern	Abstract Factory	Builder	Factory Method	Prototype	Singleton
Abstract Factory	0.00	0.18	0.10	0.45	0.22
Builder		0.00	0.28	0.43	0.18
Factory Method			0.00	0.39	0.26
Prototype				0.00	0.41
Singleton					0.00

Table 3.10: s-values between the creational patterns

Pattern	Adapter	Bridge	Composite	Decorator	Facade	Flyweight	Proxy
Adapter	0.00	0.64	0.33	0.44	0.65	0.37	0.22
Bridge		0.00	0.84	0.57	0.18	0.53	0.54
Composite			0.00	0.27	0.66	0.33	0.39
Decorator				0.00	0.45	0.27	0.58
Facade					0.00	0.36	0.36
Flyweight						0.00	0.47
Proxy							0.00

Table 3.11: s-values between the structural patterns

techniques, a certain level of human intervention is almost inevitable, and actually helpful for users; because of various uncertain aspects and limitations of the current methods.

Pattern	Chain of Responsibility	Command	Interpreter	Iterator	Mediator	Memento	Observer	State	Strategy	Template Method	Visitor
Chain of Responsibility	0.00	0.36	0.26	0.32	0.26	0.45	0.25	0.26	0.32	0.26	0.42
Command		0.00	0.21	0.04	0.23	0.28	0.14	0.27	0.06	0.23	0.42
Interpreter			0.00	0.21	0.04	0.36	0.12	0.10	0.20	0.18	0.24
Iterator				0.00	0.21	0.27	0.16	0.25	0.08	0.19	0.38
Mediator					0.00	0.32	0.12	0.11	0.21	0.14	0.20
Memento						0.00	0.31	0.39	0.28	0.20	0.27
Observer							0.00	0.17	0.18	0.16	0.30
State								0.00	0.23	0.19	0.21
Strategy									0.00	0.23	0.38
Template Method										0.00	0.30
Visitor											0.00

Table 3.12: s-values between the behavioural patterns

Having produced s-values, the accuracy of detected pattern instances can be checked by manually inspecting them by looking at the source code and/or reverse engineering UML diagrams with OO CASE tools like Rational Rose[6]. Figure 3.10 shows an example of reverse engineering class diagrams from source code using Rational Rose.

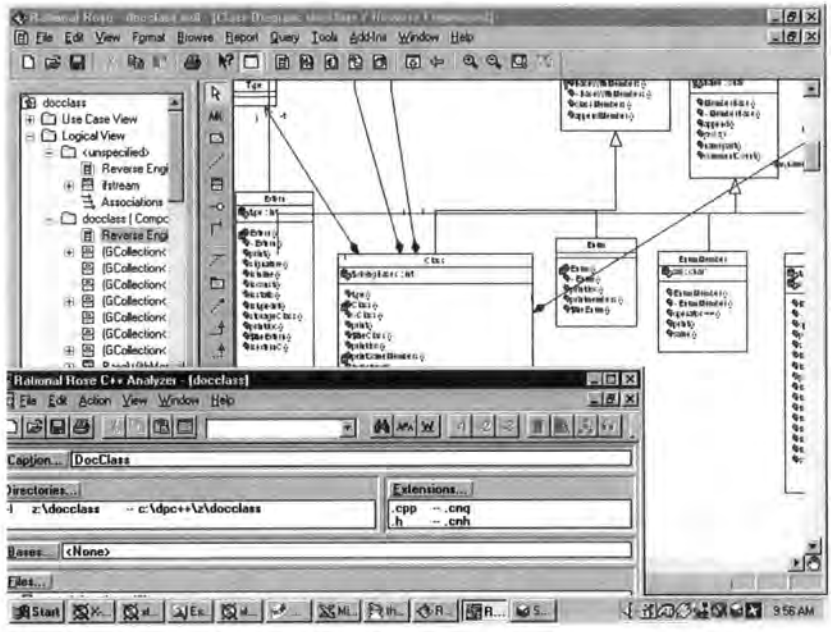

Figure 3.10: Reverse engineering a class diagram from C++ source code using Rational Rose

[6]*Rational Rose* is a registered trademark of *Rational Rose Corporation*.

3.4 Reconciliation Process after Design Pattern Recovery

In the previous section, a method was developed so that design pattern candidates can be detected by collecting various software metrics and analysing them. Having detected patterns, it is necessary to perform a reconciliation process where users verify the detected patterns, and decide what to do about them.

According to the study carried out by Shull et al., detected patterns fall into 4 different types [119]. They are *Types 1* to *4* divided by their completeness in terms of *implementation* and *purpose*. Type 4 patterns are the most desirable ones as they are complete both in their original purposes and implementations. The next desirable ones, i.e., Type 3 patterns, succeed in achieving the same purposes but they fail in their implementations. Patterns belonging to Type 2 lack in addressing their purposes but they still have sophisticated implementations. The least desirable ones belong to Type 1. They hardly address their purposes nor their implementations. Patterns belonging to Types 1, 2, and 3 are the ones that need to be reengineered for the sake of future maintenance and evolution. The reason for this is that they are hard to understand and less maintainable. Figure 3.11 shows the four different types of design patterns on two-dimensional surface with "X" and "Y" representing "Purpose" and "Implementation", respectively.

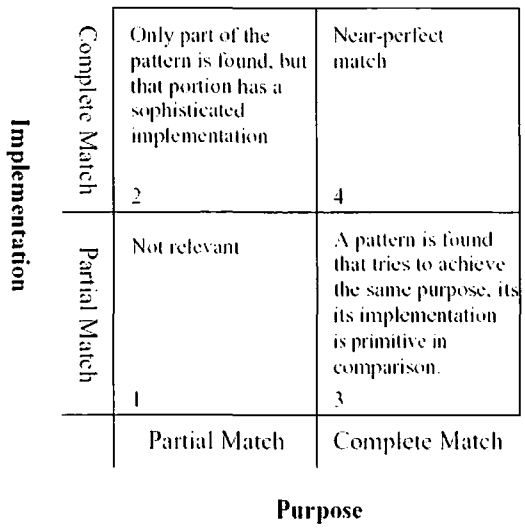

Figure 3.11: The 4 types of the recovered patterns (from Shull et al. [119])

3.5 Summary

In this chapter, the OO software development and maintenance model and the evolution of classes were studied. Then, it was followed by development of a GQM plan based on Basili's GQM method. The plan is the basis of the design pattern recovery method and is also used to evaluate the accuracy and effectiveness of the method. Software product metrics were extensively used to investigate the characteristics of each class and interactions between them. After analysing the metrics data of GoF patterns, pattern signatures were extracted. By these signatures each pattern can be distinctively identified from others.

Chapter 4

The Pattern-Based Redocumentation (PBR) Method

This chapter explains a method to document design patterns and then redocument a system using detected patterns. There is a need for good documentation during development and maintenance. This chapter proposes an approach to improving the existing documentation. This approach recovers design and architectural information, then redocuments the program by reusing the information. The PBR method is based on the XML technologies like DTD, XSL, XLink and XPointer.

4.1 Introduction

As many software engineers have confessed, software maintenance is the most time-consuming and costly activity during the software life cycle. Further, maintenance is hampered by comprehension tasks [141]. Thus applying good documentation schemes can be a promising starting point towards successful maintenance and evolution.

The documents associated with a software system have a number of requirements [123]. First, they should act as a communication medium between members of the development team. Second, they should be a system information repository to be used by maintenance engineers. Third, they should provide information for management to help them plan, budget and schedule the software development process. Finally, some of the documents should tell users how to use and administer the system. For software engineers, the first two are most relevant, leaving the last two for managers and end-users.

Most of time developers only have unreliable documentation that is often outdated, inconsistent with the other parts of the system, and difficult to comprehend. Therefore, it is necessary to overcome this situation by adopting reverse engineering and redocumentation techniques for future maintenance. In this respect, redocumentation can be viewed as a preventive maintenance activity.

Redocumentation is one of the oldest forms of reverse engineering [136]. The standard definition of the term "redocumentation" made by *the IEEE-CS Technical Council*

on Software Engineering (TCSE) - Committee on Reverse Engineering [32] reads:

> a form of restructuring where the resulting semantically-equivalent representation is an alternate view intended for a human audience.

According to the definition, redocumentation allows users to get the right understanding that reflects the human-oriented representation of their software.

There certainly exists an analogy between software development and software documentation within the software engineering community. Developers often fail in their projects because they apply the same approaches to systems of different sizes and complexities. Some principles that work well on a small scale often cannot be directly applied to large projects without some sort of modification. Therefore it is true that "documentation-in-the-large (DitL)" is as different from "documentation-in-the-small (DitS)" as "programming-in-the-large (PitL)" is to "programming-in-the-small (PitS)" [120, 135]. In PitL, a more rigorous manner of applying processes and system modelling techniques is required, not to mention a formal measurement plan like the GQM plan in order to minimise the high risks associated with large projects. In a similar manner, DitL should help users understand the whole picture of software not just localised and limited information such as data structures and algorithms.

When maintaining existing software, the need for understanding the software arises. However, most existing documentation fails to supply maintainers with enough accurate information showing different system perspectives, thus high maintenance costs

are incurred. Many times, maintainers are bombarded with tediously detailed information that is not relevant to what they are trying to do. The other times, they only receive documentation that describes the whole system architecture but without giving any detailed description of the software components. Thus it is important to strike the right balance between these two kinds of information when documenting systems [145].

"Abstraction" and "structuring" are two of the most powerful conceptual tools that software engineers can use [82]. Looking back the developments made in Computer Science, many fine examples of these can be found, not just in the software industry but also in the hardware industry. Design patterns are a good abstraction tool because they are conceptually higher than basic software building blocks like classes and objects, or data structures and algorithms. Figure 4.1 shows various software abstractions available to software engineers at present. The items at the lower parts can be used to implement or realise the higher parts ones. For example, frameworks can be designed on the basis of patterns and pattern languages that are again supported by building blocks situated at the lower abstraction levels.

The difficulties of developing good quality documentation are often compared with those of developing software itself [135]. Therefore software documentation should be treated with the same importance as software development itself.

Most software development organisations spend a substantial amount of time developing documents, and in many cases the documentation process itself is quite ineffi-

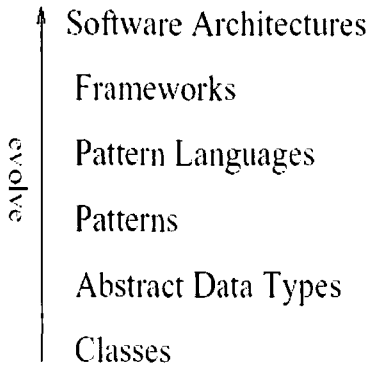

Figure 4.1: Evolution of abstractions

cient. It is not unusual for a software development organisation to spend as much as 20 or 30 percent of all software development effort on documentation [123].

Unfortunately, it has been observed that software documentation is not satisfactory when it is desperately needed. For example, documentation of software processes and products is usually neglected during software development on account of schedule pressure and budgetary constraints. Later during maintenance, both internal code documentation and external documentation tend to become inconsistent as changes are continuously made to the original software; later forcing maintainers to investigate the source code line by line in order to get an understanding of software.

4.2 Redocumenting Software Systems Using Patterns and XML: the PBR Method

As mentioned earlier, *structuring* is an indispensable conceptual tool for software engineers along with *abstraction* [82]. By dividing software into well-structured modules or subsystems, and explaining the software using abstractions captured through reverse engineering techniques, it can be expected to gain a more comprehensive understanding of the software.

Design patterns are useful for both structuring and abstraction because they are essentially collaborating classes and objects to solve recurring problems. Design patterns are conceptually higher than the usual building blocks of OOP. Rather than communicating with each other using primitive fine-grained programming features, users can convey their intentions to other people more quickly and correctly. Another advantage of using patterns as a documentation tool is the reduction of mistakes. This is because time-tested conceptual components are used. A recurring structure is considered as a valid pattern only if it appears in several different applications [45].

In this research various XML technologies are utilised to represent data in a consistent manner by way of DTD. For the display of the information XSL is applied to XML documents. This scheme gives users a more flexible documentation structure than when using normal text formats or HTML. This results from the fact that in XML

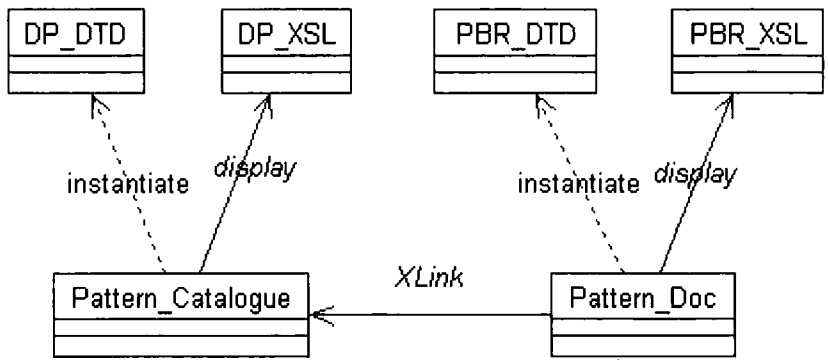

Figure 4.2: Redocumenting a system using patterns and XML technologies

meaningful tag names can be assigned, and data or content as an abstraction is separated from any presentation information. Key elements of the PBR method are shown in Figure 4.2.

4.2.1 Consistent Representation of XML Documents

Two DTDs are developed and XML documents are validated against these[1]. This will be useful for ensuring the consistency between XML documents. These documents contain pattern instances detected from a system. A pattern catalogue consisting of the 23 different GoF patterns is documented. The size of this pattern catalogue will certainly increase as users find more patterns. Each element of the first DTD roughly matches up with the sections of the pattern template. By marking up different sections with some meaningful tags rather the simple HTML tags, the information can be used

[1] Appendix A shows the actual source code of the two DTDs and their instantiated XML documents along with their corresponding style sheets.

in a more useful manner.

If a user wants a different pattern description template, he may define the above tags differently. For example, Frank Buschmann and his collegues from Siemens developed a slightly different pattern catalogue [30]. It will be interesting to combine theirs and GoF's together in the DTD.

The concept of pattern language is still very weak so that the PBR method does not use this concept. Using a pattern language is not a particularly good idea, because the DPR method can only recover the existence of individual patterns in code.

Once patterns have been detected from a system, then the system can be redocumented using this *pattern information* along with other kinds of information like *class documentation* and *metrics documentation.* Class documentation is obtained with a CASE tool called "DocClass[2]" while metrics documentation is from "CCCC". Class documentation contains various information on classes and objects such as attributes and methods. DocClass produces this documentation based on in-line comments and source code analyses. Thus it can be said that DocClass is comparable to the *javadoc* utility available to the Java programmers.

The grammar and vocabulary of DTD are similar to those of classes of UML. Figures 4.3 and 4.4 are graphical representation of the above two DTDs. These were

[2]DocClass is a simple C++ program which reads in C++ header files, and outputs documentation describing the class hierarchy, methods, inherited methods, etc. It was developed by Trumphurst Ltd.

reverse engineered using Rational Rose's XML DTD reverse engineering facility to get the overview of the different elements and attributes existing in the two DTDs. In these figures, each class represents their corresponding elements and entities in the two DTDs, while attributes of each XML element are transformed into attributes of their matching classes. Each class in linked with other classes according to their composition and groupings. These are association relationships and represented with *directed arrows* in these class diagrams.

Because the XML documents used in the PBR method mainly consist of texts and UML diagrams, DTD was chosen rather than the more complex and powerful XML Schema.

The metrics documentation contains various software metrics. In fact these metrics were used to implement the DPR method in Chapter 3.

Finally, the pattern documentation includes names of patterns and their c-classes classes. Of course, users can change the structure of these DTDs as they see fit to whatever their different situations require.

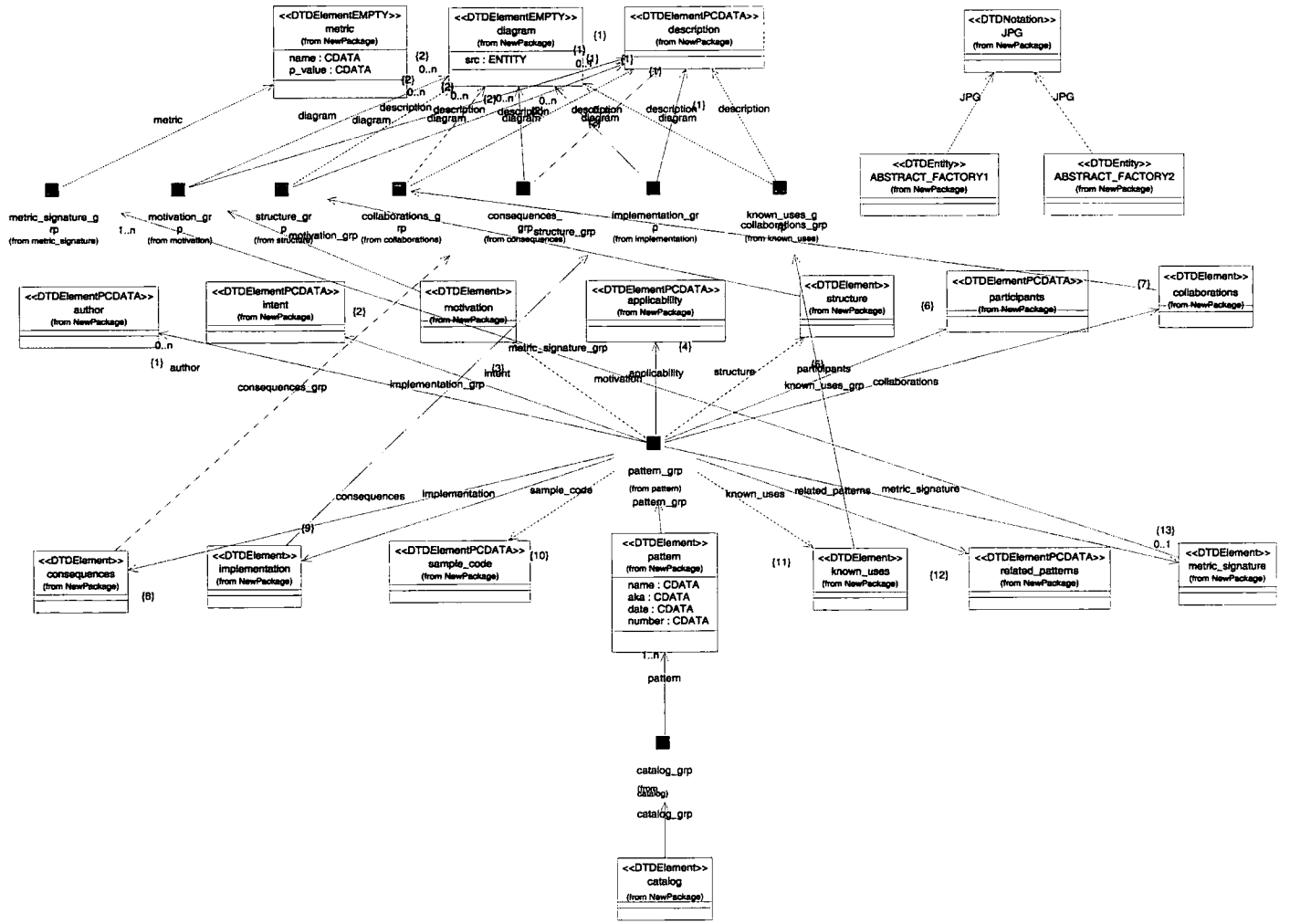

Figure 4.3: A class diagram reverse engineered from pattern.dtd

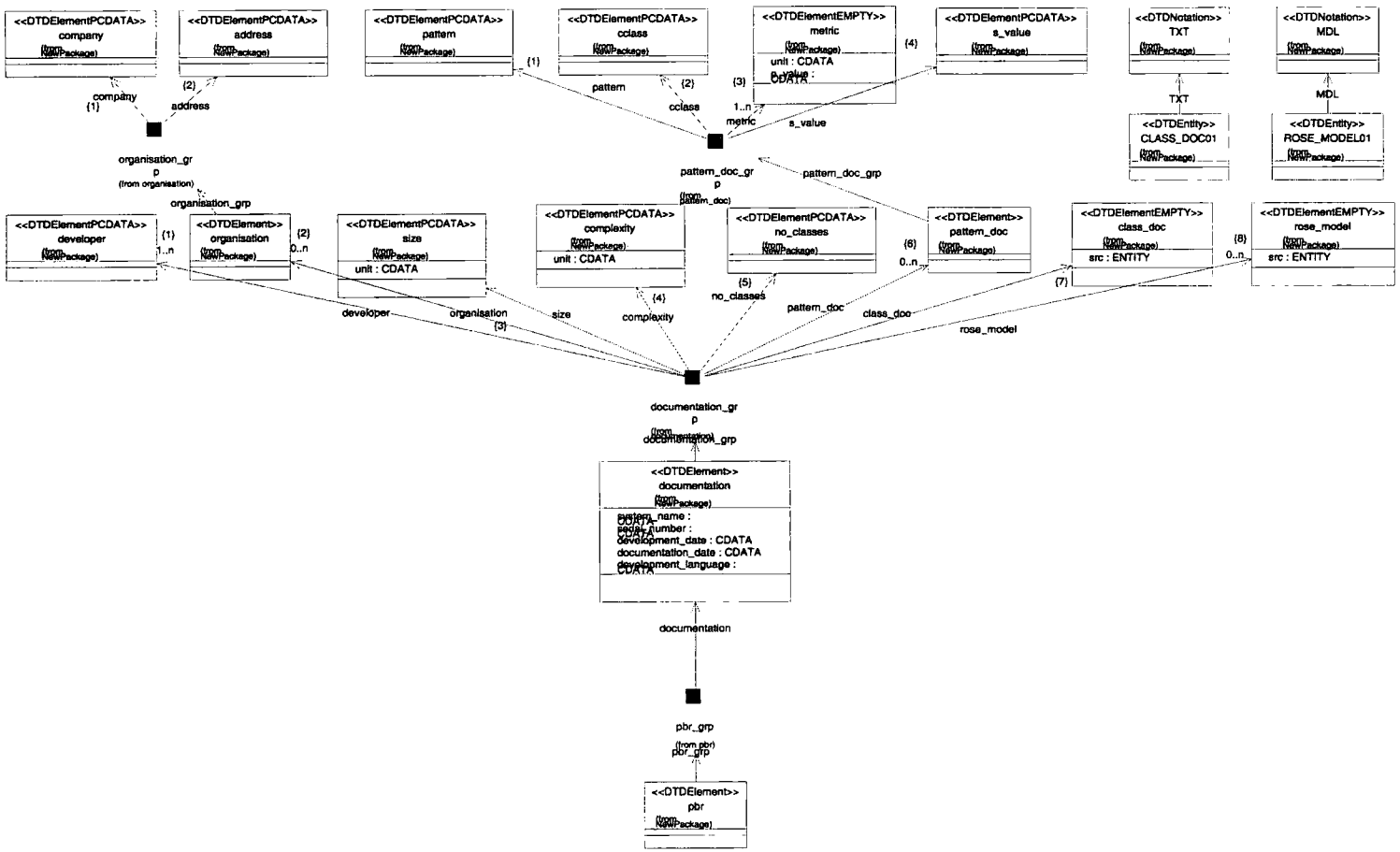

Figure 4.4: A class diagram reverse engineered from pbr.dtd

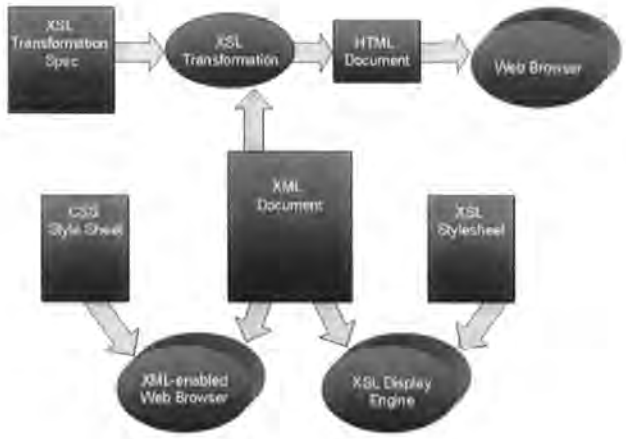

Figure 4.5: Options for displaying XML documents

4.2.2 Flexible Representation of XML Documents

Unlike HTML, XML does not contain any information regarding how its different elements can be viewed. There are many different ways of displaying an XML document on a web browser. Figure 4.5 shows the three different ways of displaying an XML document on a web browser. At the time of writing this thesis, there are only few web browsers that are capable of displaying XML documents including Microsoft Internet Explorer, Netscape Communicator and InDelv[3]. In this research, InDelv was chosen because it supports formatting objects (FOs) and XLink unlike the other two.

People used many various formats to describe patterns, e.g., plain text or HTML in the case of the electronic version of the design pattern book. However, it is difficult to describe patterns using those formats. Also managing those types of documents can

[3] *Internet Explorer*, *Netscape Communicator*, and *InDelv* are registered trademarks of *Microsoft Corporation*, *Netscape Communications Corporation*, and *InDelv Inc.*, respectively.

be as difficult as maintaining software systems. As for descriptions written in HTML, HTML tags are not descriptive, not having any semantic meanings. Because of this, they are difficult to use; although they are good as a representation form on a web browser.

Two XSL documents were developed to display the two kinds of XML documents on InDelv. Figures 4.6 and 4.7 are examples of applying these two XSL documents to their related XML documents.

There are three different kinds of documentation available in a PBR XML document, i.e., class, metrics and pattern documentation. Upon selecting one of these, users are directed to the respective documentation. Figures 4.8 and 4.9 show the first two kinds of documentation that users get.

4.3 Summary

Recovering patterns will only be meaningful and useful when they can be applied to help software engineers perform their various software engineering activities. In this chapter, one application of patterns, i.e., redocumenting software was discussed. This PBR method utilises the information of the detected patterns in a system along with other kinds of documentation like class and metrics documentation. This method was implemented using some of the XML technologies such as DTD, XSL and XLink. It can be claimed that the PBR method can help users produce and maintain system

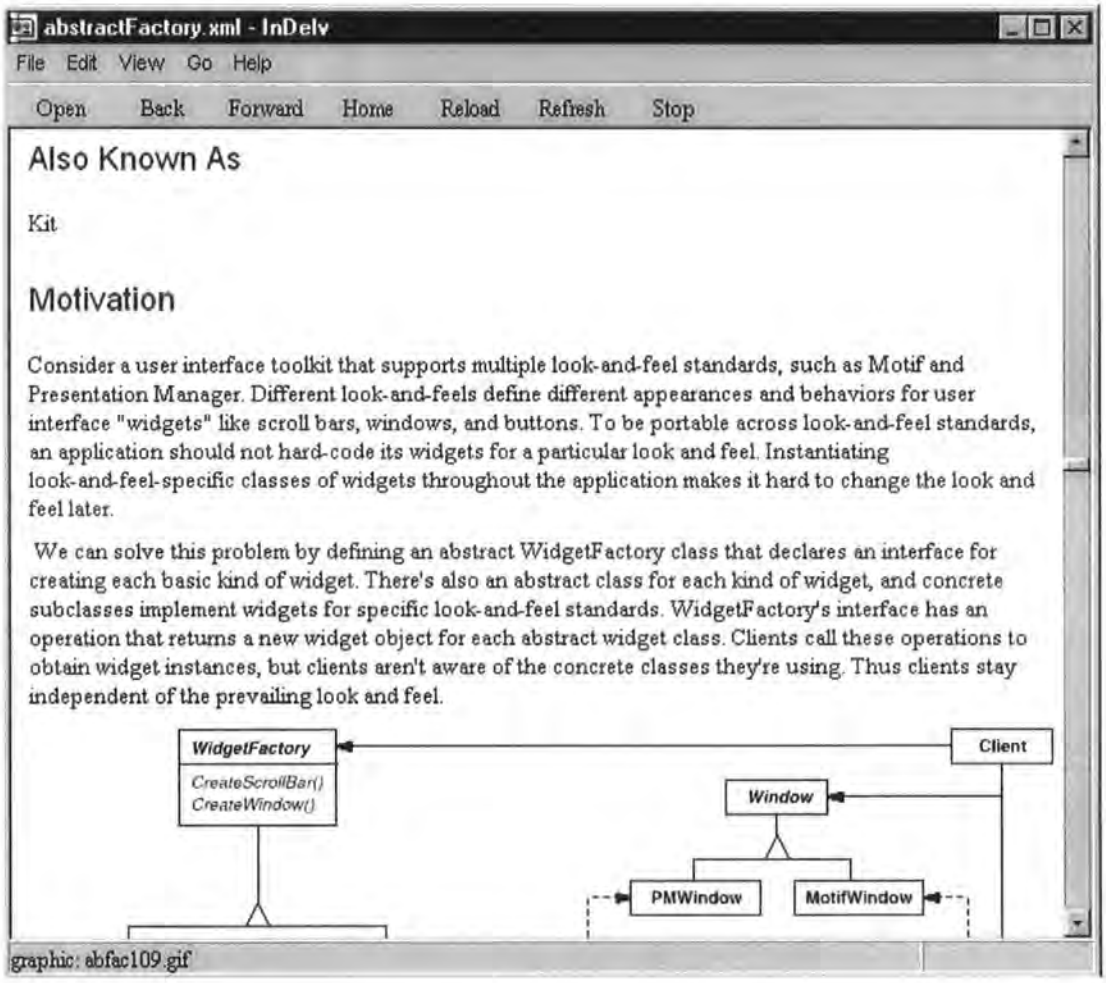

Figure 4.6: The rendering of a pattern description in XML using XSL

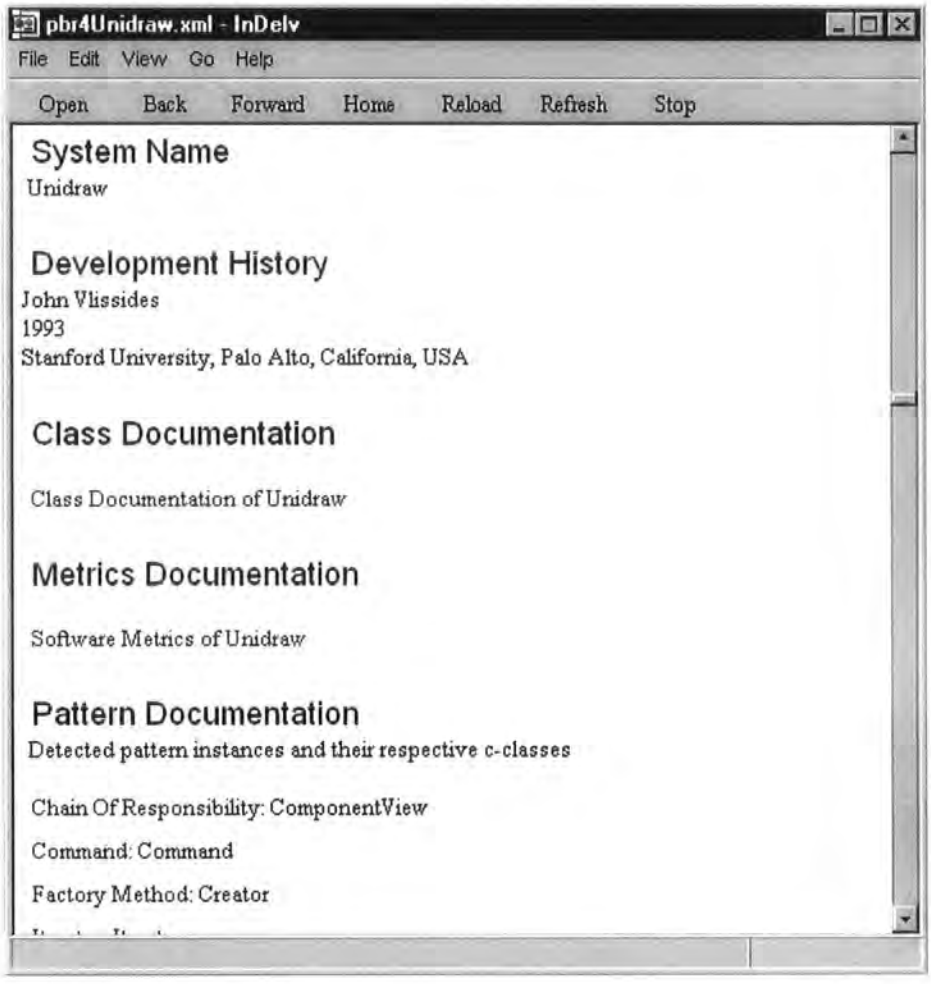

Figure 4.7: The rendering of a PBR description in XML using XSL

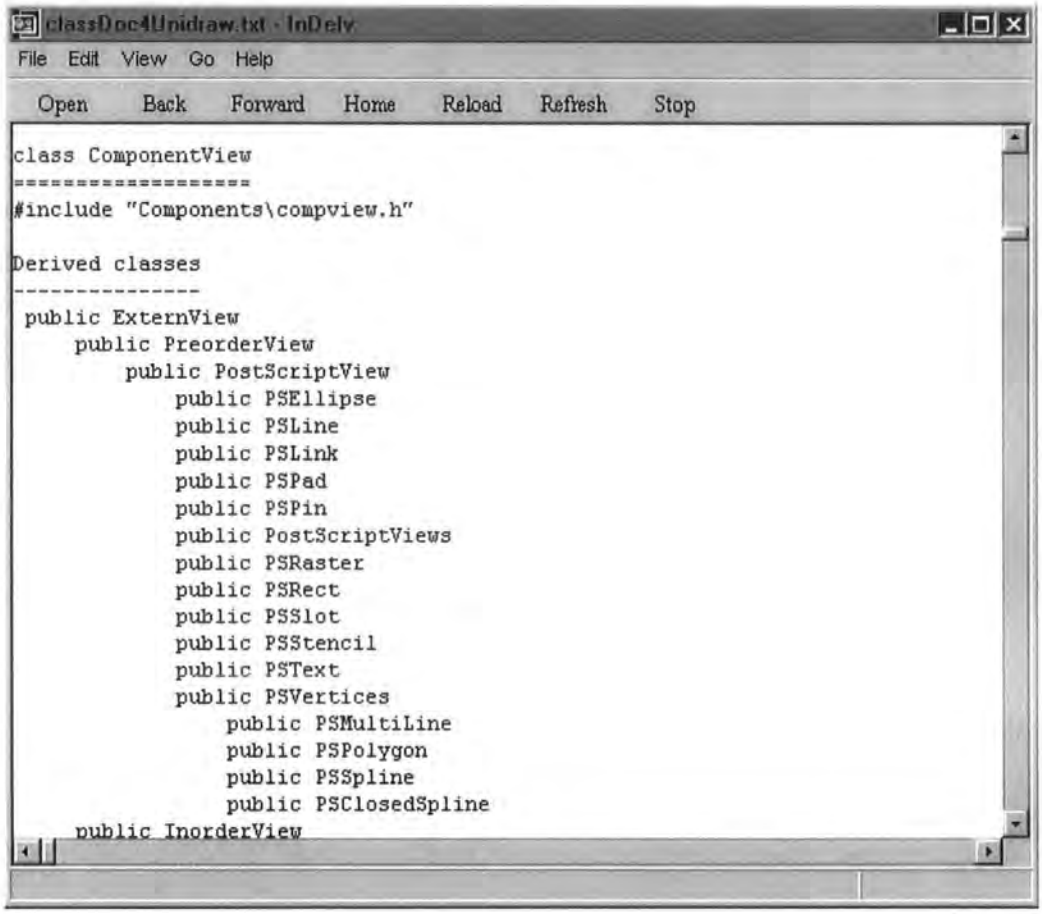

Figure 4.8: The access to class documentation from a PBR XML document

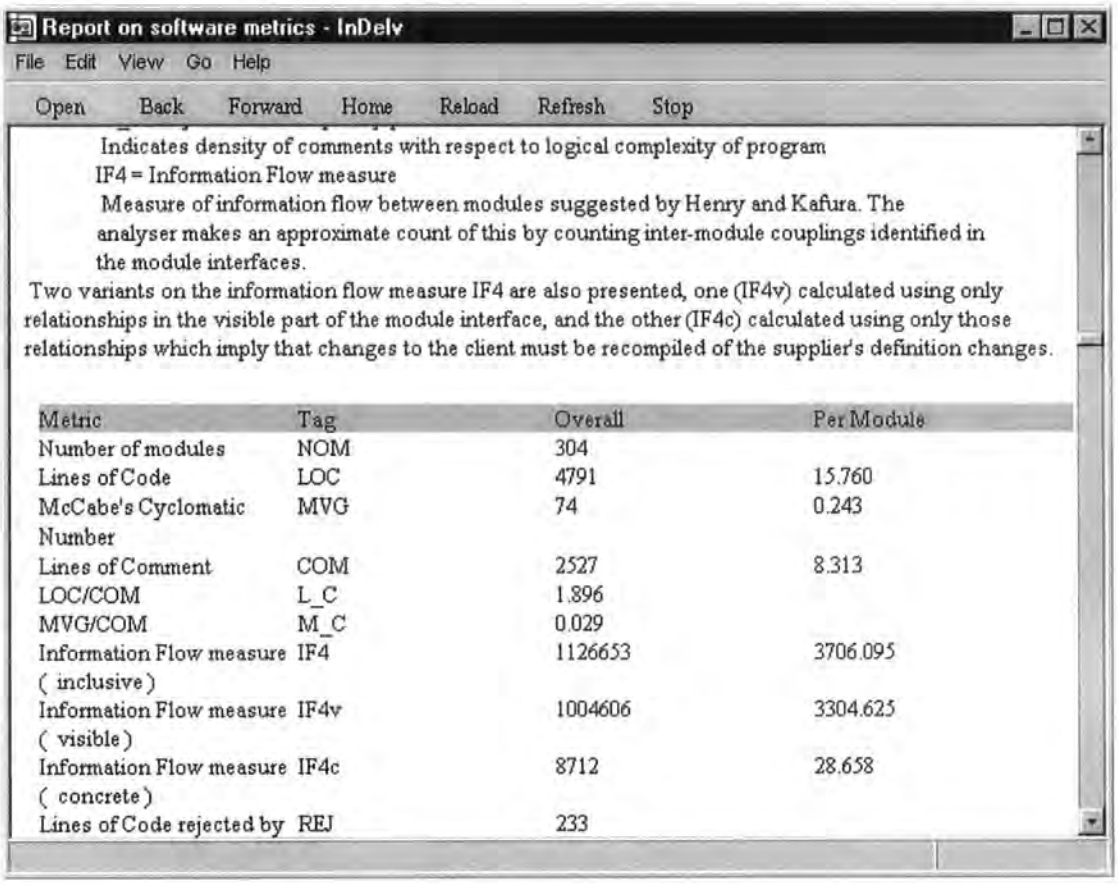

Figure 4.9: The access to metrics documentation from a PBR XML document

documentation. By doing so, the problems of software maintenance can be eased as well because the two things are very much associated with each other.

Sometimes redocumenting software is not enough, and it might be necessary to carry out more serious program restructuring tasks. However, these restructuring and reengineering of systems were not covered in this research.

Chapter 5

Case Study

In this chapter, experiments with some systems within which design patterns have previously been identified. They are documented in the design pattern book and they are ET++, Unidraw and two different versions of InterViews, i.e., 2.6 and 3.2a. Using these systems as experimental materials is better and fairer than manual checking of the trueness of patterns appearing in a system because manual checking is difficult, if not impossible with large systems, and can incur errors. A more objective and correct evaluation of the detection method can be guaranteed in the former case. Through these experiments, the usefulness and limitations of the approach can be judged.

5.1 Introduction

In Chapter 3, the DPR method was developed. Experiments are needed to see whether this method is suitable and sound.

In the rest of the chapter, first the experimental framework is described. Then, the experimental results are reported. Finally, a detailed experimental analysis of the results is given.

5.2 Experimental Framework

Experimental Goals

The main goal of the experiments is to investigate the usefulness and correctness of the pattern recovery method. This goal was set out specifically when the GQM plan for this research was discussed in Chapter 3 .

Experimental Materials

As previously explained within this chapter, the four systems are used. The existence of GoF patterns in these systems are documented in a detailed manner. By selecting these experimental materials, the fairness and ease of the experiments can be guaranteed.

Software	LOC	MVG	No. of Classes	Year	Place
ET++ 3.0b	53202	10284	579	1993	Univ. of Zurich, Switzerland
InterViews 2.6	4400	200	215	1989	Stanford Univ., USA
InterViews 3.2a	8667	200	507	1993	Stanford Univ., USA
Unidraw	4791	74	304	1993	Stanford Univ., USA

Table 5.1: The experimental materials

Table 5.1 is a brief overview of these systems with respect to their sizes, complexities, development years and organisations.

The following is a brief introduction to these systems.

ET++ is a portable and homogeneous object-oriented class library integrating user interface building blocks, basic data structures, and high level application framework components [144]. ET++ eases the building of highly interactive applications with consistent user interfaces following the direct manipulation principle. A byproduct of the ET++ project is a set of tools, which were designed to support the exploration of ET++ applications at run-time. The ET++ class library is implemented in C++ and can be used on several operating systems and window system platforms. Since its initial conception the class library has been continuously redesigned and improved. It originated from an architecture which was close to **MacApp** [4]. During several iterations a new and unique architecture evolved.

InterViews is an object-oriented user interface package that supports the composi-

tion of a graphical user interface from a set of interactive objects [74, 73]. The base class for interactive objects, called an *interactor*, and base class for composite objects, called a *scene*, define a protocol for combining interactive behaviours. Subclasses of scene define common types of composition: a box tiles its components, a tray allows components to overlap or constrain each other's placement, a deck stacks its components so that only one is visible, a frame adds a border, and a viewport shows part of a component. Predefined components include menus, scrollers, buttons, and text editors. InterViews also includes classes for structured text and graphics. InterViews is written in C++ and runs on top of the X window system. Here two different versions of this software, i.e., Versions 2.6 and 3.2a are selected. As the software evolved, slightly different design concepts were introduced, thus adding new design patterns.

Finally, **Unidraw** is a framework for creating graphical editors in domains such as technical and artistic drawing, music composition, and circuit design [139]. The Unidraw architecture simplifies the construction of these editors by providing programming abstractions that are common across domains. Unidraw defines four basic abstractions: *components* encapsulate the appearance and behavior of objects, *tools* support direct manipulation of components, *commands* define operations on components, and *external representations* define the mapping between components and the file format generated by the editor. Unidraw also supports multiple views, graphical connectivity, and dataflow between components. As from InterViews Version 3.1 released in 1993, Unidraw has been included in InterViews. Unidraw was the basis of

John Vlissides' thesis work at Stanford University [138]. He is one of the four authors of the design pattern book.

Experimental Method

In Chapter 3, pattern signatures were extracted on the basis of a GQM plan. Those signatures are used to check whether there is a strong relationship between patterns of metrics and patterns themselves. It is desirable to perform experiments with systems where the usage of patterns is known. In order to map the signatures to classes in the subject systems, first the c-classes of the patterns present in those four systems were identified. These c-classes were identified by analysing their descriptions appearing in the pattern book. Then, their metrics are obtained, and their p-values are computed. Finally, these p-values are compared with the p-values of each pattern signature to get the s-values. By checking these s-values, the correctness of the pattern recovery method is judged. In other words, if s-values are reasonably low, then it proves that the method works well.

5.3 Experimental Results and Their Analysis

By following those steps described in the previous section, experimental results were produced. Tables 5.2, 5.3, 5.4, and 5.5 show the patterns present in each system, their c-classes and s-values.

Pattern	c-class	s-value
Abstract Factory	WindowSystem	0.25
Bridge	WindowPort	0.89
Builder	Converter	0.28
Chain of Responsibility	EvHandler	0.19
Composite	Vobject	0.11
Command	Command	0.46
Decorator	Stream	0.26
Facade	EtProgEnv	0.29
Factory Method	Application	0.41
Flyweight	Layout	0.27
Iterator	Iterator	0.38
Observer	View	0.49
Proxy	ImageCache	0.47

Table 5.2: Patterns in ET++ and their c-classes and s-values

Pattern	c-class	s-value
Adapter	GraphicBlock	0.22
Composite	Graphic	0.11

Table 5.3: Patterns in InterViews 2.6 and their c-classes and s-values

Pattern	c-class	s-value
Abstract Factory	WidgetKit	0.26
Abstract Factory	DialogKit	0.29
Abstract Factory	LayoutKit	0.43
Adapter	GraphicBlock	0.35
Command	Action	0.30
Composite	Graphic	0.15
Decorator	DebugGlyph	0.48
Flyweight	Glyph	0.33
Observer	Observer	0.10
Singleton	Session	0.41
Strategy	Compositor	0.24

Table 5.4: Patterns in InterViews 3.2a and their c-classes and s-values

Pattern	c-class	s-value
Chain of Responsibility	ComponentView	0.19
Command	Command	0.43
Factory Method	Creator	0.55
Iterator	Iterator	0.22
Mediator	Connector	0.29
Memento	MoveCmd	0.23
Observer	ComponentView	0.18
Prototype	GraphicCompTool	0.10
State	Tool	0.14

Table 5.5: Patterns in Unidraw and their c-classes and s-values

A total of 35 pattern instances were present in those four systems and they are of 20 different kinds of patterns. Remembering that the total number of GoF patterns is 23, it is right to say that these instances cover most cases.

The sum of the s-values of the 35 pattern instances is "10.75" and their mean, "0.3071". This mean value is used as a critical point by which positive cases and negative cases are divided. In other words, a case is positive if its s-value is less than "30.00". For instance, the class *Observer* of InterViews 3.2a is a highly positive case of being an Observer pattern as its s-value is just "0.10".

Among the s-values of the 35 c-classes, 60%, i.e., 21 cases are less than 0.30 evi-

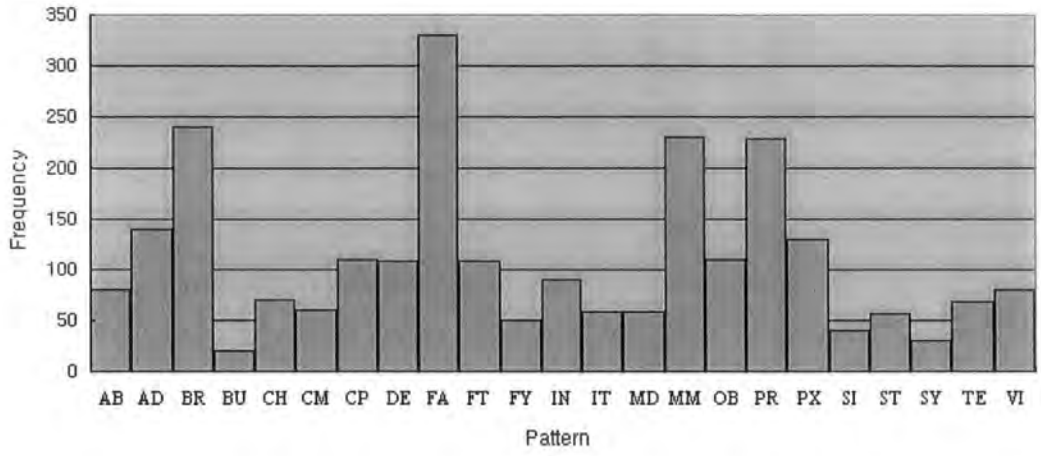

Figure 5.1: Positive false cases obtained with ET++

dencing that there exists a mapping between design patterns and their metrics patterns.

In the case of false cases, i.e., picking out instances that are not really patterns, Figures 5.1, 5.2, 5.3, and 5.4 show the positive false cases detected from the four systems, respectively. As shown in the figures, the DPR method has some weakness in the case of the Bridge, Facade, Memento, Prototype, and Proxy patterns. The method detected those instances as patterns but they were not really.

Tables 5.6, 5.7, 5.8, and 5.9 summarise the above findings. The values in the third column are the proportion of different positive or negative cases to the overall true or false cases, accordingly. For example, the proportion of the positive true cases detected from ET++ is 7/(7+6) = 0.5385 as the frequencies of the positive true cases and negative true cases are 7 and 6, respectively.

In addition to the above assessment, the correlations between the correctness of

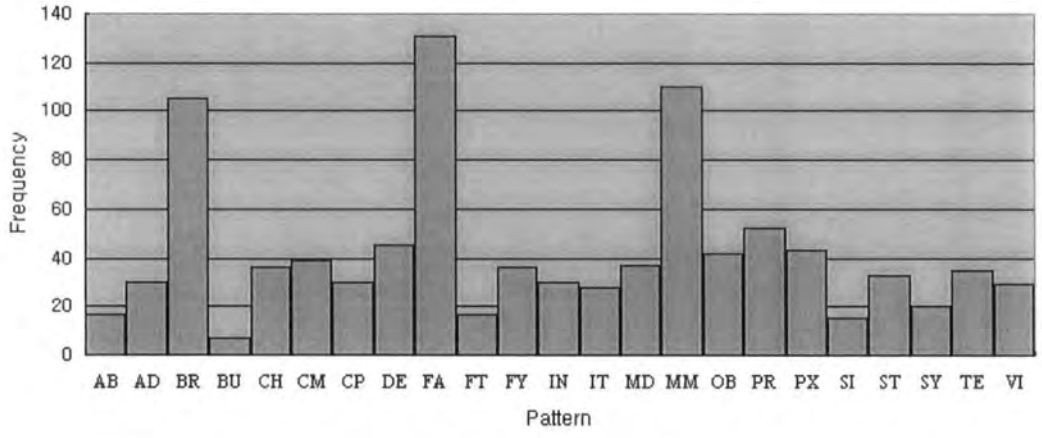

Figure 5.2: Positive false cases obtained with InterViews 2.6

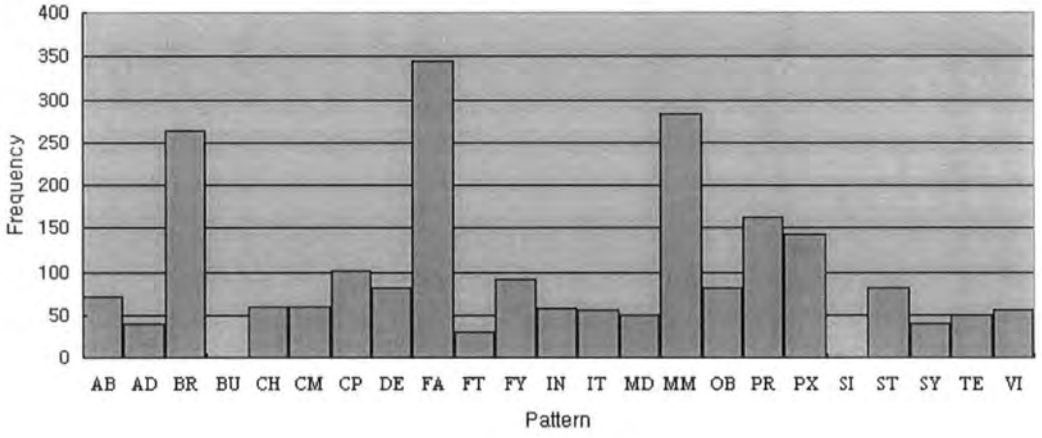

Figure 5.3: Positive false cases obtained with InterViews 3.2a

Case	Frequency	Proportion
Positive True	7	53.85%
Negative True	6	46.15%
Positive False	2498	18.78%
Negative False	10806	81.22%

Table 5.6: Different cases obtained with ET++

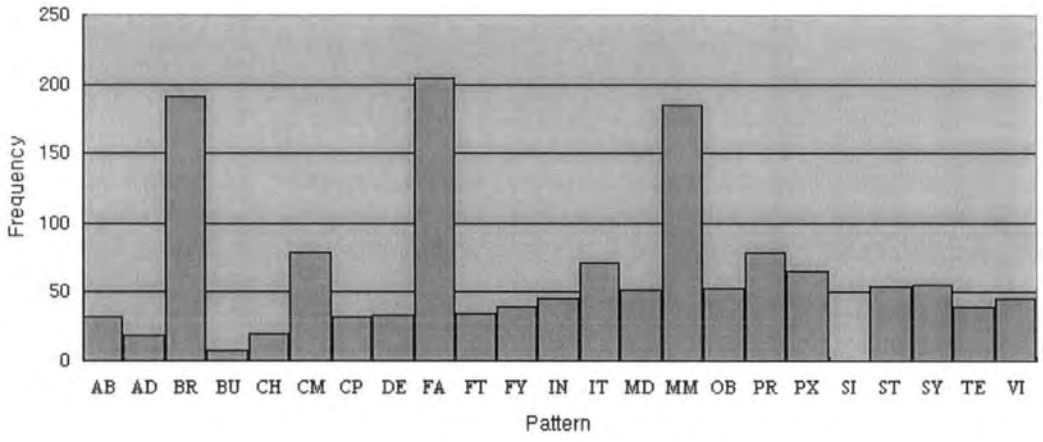

Figure 5.4: Positive false cases obtained with Unidraw

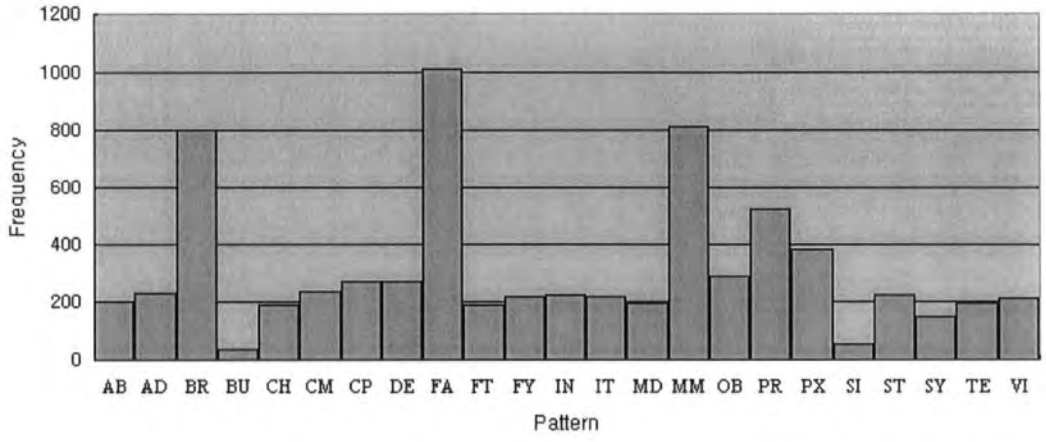

Figure 5.5: Accumulation of positive false cases obtained with the four systems

Case	Frequency	Proportion
Positive True	2	100.00%
Negative True	0	0.00%
Positive False	967	19.56%
Negative False	3976	80.44%

Table 5.7: Different cases obtained with InterViews 2.6

Case	Frequency	Proportion
Positive True	5	45.45%
Negative True	6	54.55%
Positive False	2199	18.88%
Negative False	9451	81.12%

Table 5.8: Different cases obtained with InterViews 3.2a

Case	Frequency	Proportion
Positive True	7	77.78%
Negative True	2	22.22%
Positive False	1438	20.59%
Negative False	5545	79.41%

Table 5.9: Different cases obtained with Unidraw

Pattern	c-class	s-value
Abstract Factory	WindowSystem	0.25
Abstract Factory	WidgetKit	0.26
Abstract Factory	DialogKit	0.29
Abstract Factory	LayoutKit	0.43
Builder	Converter	0.28
Factory Method	Creator	0.55
Factory Method	Application	0.41
Prototype	GraphicCompTool	0.10
Singleton	Session	0.41

Table 5.10: Instances of creational patterns and their s-values

the detection method and the category of patterns, i.e., creational, structural and behavioural pattern categories are examined. This will help reveal any limitations and weakness of the method. Tables 5.10, 5.11, and 5.12 show the s-values of the patterns belonging the three different categories.

According to the results, the accuracy of detection is more or less same irrespective of the different pattern categories. In the case of the false cases, however, the method is less accurate when dealing with structural patterns than the other two categories of patterns. This is due to the fact that the DPR method has weakness in processing such structural patterns as the Bridge, Facade, and Proxy patterns as mentioned earlier in this section. This may indicate that the software metrics used in the recovery method

Pattern	c-class	s-value
Adapter	GraphicBlock	0.22
Adapter	GraphicBlock	0.35
Bridge	WindowPort	0.89
Composite	Vobject	0.11
Composite	Graphic	0.11
Composite	Graphic	0.15
Decorator	Stream	0.26
Decorator	DebugGlyph	0.48
Facade	EtProgEnv	0.29
Flyweight	Layout	0.27
Flyweight	Glyph	0.33
Proxy	ImageCache	0.47

Table 5.11: Instances of structural patterns and their s-values

Pattern	c-class	s-value
Chain of Responsibility	EvtHandler	0.19
Chain of Responsibility	ComponentView	0.19
Command	Command	0.46
Command	Action	0.30
Command	Command	0.43
Iterator	Iterator	0.38
Iterator	Iterator	0.22
Mediator	Connector	0.29
Memento	MoveCmd	0.23
Observer	View	0.49
Observer	Observer	0.10
Observer	ComponentView	0.18
State	Tool	0.14
Strategy	Compositor	0.24

Table 5.12: Instances of behavioural patterns and their s-values

Case	Frequency	Proportion
Positive True	5	55.56%
Negative True	4	44.44%
Positive False	1001	12.49%
Negative False	7015	87.51%

Table 5.13: Different cases obtained with the creational patterns

Case	Frequency	Proportion
Positive True	7	58.33%
Negative True	5	41.67%
Positive False	3177	28.31%
Negative False	8046	71.69%

Table 5.14: Different cases obtained with the structural patterns

have some limitations to represent the information contained in those patterns, whether it is semantic and/or syntactic dynamic information. Tables 5.13, 5.14, and 5.15 show how the method works with the different categories of patterns.

5.4 Summary

In this chapter, case studies were performed to evaluate the recovery method using the four pieces of software. The systems have patterns in them and their existence

Case	Frequency	Proportion
Positive True	9	64.29%
Negative True	5	35.71%
Positive False	2924	16.58%
Negative False	14717	83.42%

Table 5.15: Different cases obtained with the behavioural patterns

was well documented in the literature, so that they were good materials for the experiments. According to the results produced, the method has a reasonably high degree of accuracy although depending on the pattern categories, the accuracy varies in some cases.

Chapter 6

Conclusions

This chapter summarises the research reported in this thesis. First, a list of the major contributions made in this research is given and these contributions are evaluated according to the success criteria set out in Chapter 1. Then, some areas of research requiring further investigation are suggested. Finally, this chapter ends with some concluding remarks.

6.1 Summary of Contributions and Their Evaluation

At the start of this thesis, two research problems that the software engineering community is currently facing were identified. One is the lack of design reuse and another is the immaturity of software measurement practices.

A survey of the literature on software reuse in general and software design reuse in particular has been presented. Reuse is considered to be a partial solution to the software crisis. The limitations of the more traditional code-based reuse have been presented, and thereafter existing promising design reuse approaches were surveyed. Software architecture, OO frameworks, and design patterns were recognised as currently important and popular solutions, having attracted the interests both from the industry and the academia. Finally, design patterns were chosen for the main reuse approach to be adopted in this research because of their scale and abstraction level.

In the following two chapters, a detailed explanation of the DPR method and PBR method is given. This approach is meaningful in that it first recovers design information in the form of patterns, then supports their reuse to redocument existing systems. The two methods were realised using existing tools like MS Excel 97 and an XML-enabled browser, InDelv. In the DPR method, two new metrics, i.e., *p-value* and *s-value* were defined. P-value is computed based on the ranking and percentile of each metric, while s-value is a similarity metric computed by applying different weights to each p-value.

A case study was carried out to see whether this approach was able to detect documented patterns in existing systems. Four applications were used in the case study. They varied in their sizes, development environments, and the number and kinds of patterns that they contain. Due to the fact that the existence of patterns in these systems is known, the correctness and fairness of these experiments can be ensured. According

to the experimental results, it was observed that there is a strong relationship between metrics of patterns and patterns themselves; although the method shows some weakness in detecting such patterns as the Bridge, Facade, Memento, Prototype, and Proxy patterns. These patterns are difficult to be detected using the metrics-based approach because of their structural and behavioural characteristics.

In Chapter 1, four criteria for success of this research were set out. Below an explanation of how those criteria were satisfied during the course of this research is given.

The first criterion was developing a design pattern recovery method that can recover patterns semi-automatically. The DPR method used the three kinds of software product metrics, i.e., procedural, structural, and object-oriented metrics, thus matching those programming paradigms where an typical OO system is built.

Unlike other design pattern recovery methods and more traditional design recovery methods, this method is purely based on software measurement. The method utilises the object-oriented software development model and the GQM method. The object-oriented programming paradigm evolved from the structural programming paradigm, that was in turn developed to complement the weakness of the earlier procedural programming paradigm. Thus if users want to grasp both the structure and behaviour of an OO system, they need to investigate these three kinds of characteristics. Fortunately, the software measurement community has been developing software metrics that can cover these three aspects, i.e., procedural, structural, and object-oriented metrics.

Although the method was only applied to C++ programs, it is easily applicable to other OO programming language environments such as Ada95 and Java as the metrics used can be obtained for these languages as well.

The second evaluation criterion was about applying those patterns recovered using the DPR method to redocument programs efficiently. Among the many potential usages of patterns, the main focus of interest in this research was program redocumentation; as it increases program comprehensibility and reduces maintenance and evolution cost. This method incorporates pattern, class and metrics information into existing kinds of documentation to address *documentation-in-the-large (DitL)* as well as the usual *documentation-in-the-small (DitS)*. This is due to the fact that patterns are at a higher level of abstraction than commonly used building blocks like class and procedure.

Although a CASE tool that realises the pattern recovery and pattern redocumentation methods will be potentially useful for users, instead a decision was made to use existing tools available such as MS Excel and XML browsers. By doing so, it also increases the applicability of these methods.

The final criterion was the validation of the pattern recovery method through case studies. A case study was conducted to inspect whether the method is sound and useful. It was observed that it is easy to apply and produced some good results. Judging from the case study, it can be claimed that the DPR method is useful for recovering patterns

without need for significant user intervention.

Chapter 4 discussed the application of patterns in redocumentation. However, the evaluation of the actual benefits of using patterns in redocumentation by practising engineers is beyond the scope of this research. The value of patterns has already been addressed in Chapters 1 and 2. Therefore a rationale for trying to recover patterns has been provided if they can be identified in the code.

As a whole, the methods help achieve design reuse beyond the very low source code level.

6.2 Limitations of Approach

There are some problems regarding the DPR method. These are described below.

First, the C++ language has many different dialects. Thus the method sometimes fails to produce correct results. If the Java language had been used, more correct results might have been produced as it is more standardised.

Second, to improve the correctness of the method, it is necessary to identify more metrics and collect comprehensive and controlled industrial data. Unfortunately, this was not possible during the time frame of this research.

Third, the recovery method is only useful for detecting an already known set of

patterns, i.e, GoF patterns in this research. It is difficult to recover new patterns using this method, at least without modifying and improving the pattern matching algorithm. In addition, there are some patterns that are not detected easily because of their high similarity with other patterns. This is due to the fact that they have low s-values with each other.

6.3 Other Areas for Future Research

There are some research issues that are not resolved during this research that may be worthwhile to investigate in the future.

First, it will be interesting to see whether patterns can be detected at the earlier stages during the software life cycle rather than when the implementation of a system has been finished. This would certainly improve the quality of the final products and improve the communication between the project participants. Some existing code metrics can be used for this purpose as they are. However, it is most likely that a process of adaptation and defining new metrics is required. Some examples of the metrics applicable in analysis and modelling stages have been already developed based on the UML metamodel [126, 127, 128]. This approach might produce more correct results than simply using software product metrics because design patterns can be encoded at a higher level of abstraction from that currently used. This is because the syntactic structures of patterns are quite similar to each other while their semantic information varies

greatly. When this research began, a standardised OO modelling language was not available. Now, there exists the Unified Modelling Language (UML) which is broadly embraced by the academia and the industry. By developing metrics usable onto the 6 different kinds of UML diagrams, it is possible to specify the dynamic information of patterns more precisely as well as their static information. It is this dynamic information that the DPR method lacks, and this has prevented this research from achieving a higher precision of detection.

Second, with respect to the PBR method, it provided users with a textual representation using the formatting objects of XSL. It will be worthwhile to research more sophisticated visualisation approaches of recovered design patterns, e.g., to what extent and how patterns could be visualised.

Third, there have been only a few pieces of work investigating the effects of patterns on a system that contain them. For example, some patterns can be used to restructure software to obtain the desirable properties of high cohesion and low coupling. The implications of using certain patterns against the others need to be studied more comprehensively in the future.

Fourth, designing OO frameworks is known to be extremely difficult as it requires designers to provide not only features for the current use but also ones for future usage. These future uses are difficult to predict. Reliable OO frameworks can be more easily built by gradually evolving and transforming legacy systems into pattern languages.

Pattern languages themselves are made of detected patterns from the systems. They decompose a big problem that is tackled by a software system into subproblems that subsequently are addressed by each pattern. This kind of process is good for revealing the conflicts between each module and trying to resolve them at the same time. Also pattern languages provide an overall picture of the system that is essential for designing frameworks in the first place. It is proposed that using pattern languages can ease users of the difficult tasks of building frameworks, but this remains to be comprehensively validated.

Finally, there is a potential benefit of developing more formal and efficient ways of representing pattern languages and cataloguing them. One plausible solution is through extending the UML metamodel in order to incorporate patterns directly into UML modelling elements as other researchers did for their own purposes [70, 131, 40]. By specifying patterns more formally, it will be possible to embed those pattern-related features directly in new programming languages. This could certainly improve the power of the current OOP languages.

6.4 Final Remarks

There are many important questions to be answered regarding patterns.

Do patterns stifle creativity? This same question has been asked during the course of software reuse research in the past decades. Because of their limited resources to

meet the current software development demands, developers do not have any option but to reuse the systems that they have built previously. There is a strong argument for the usefulness of patterns with respect to their educational purposes for training inexperienced developers and stimulating the ways that more experienced developers think of a good solution to a problem in a specific context.

Do people want to share their patterns with others? This is highly unlikely because the information contained in a pattern can be a crucial one for the business interests of an organisation. Except some general purpose patterns such GoF patterns, most patterns will be kept inside the organisations that discovered them. Therefore it can be predicted that domain-specific patterns will be more likely to flourish than domain-wide ones.

The patterns movement has changed the way developers build their OO systems and communicate them with others more effectively. This takes the field one step closer to the *component-based software engineering (CBSE)* originally envisioned by McIlory in the late 1960s [84].

Appendix A

PBR Templates

The following is the design pattern DTD.

```
<!-- pattern.dtd -->
<!ELEMENT pattern (heading, hr, name, classification, intent,
alsoKnownAs, motivation, applicability, structure, participants,
collaborations, consequences, implementation, sampleCode,
knownUses, relatedPatterns)>
<!ELEMENT name (title,para1)>
<!ELEMENT classification (title,para1)>
<!ELEMENT intent (title,para1+)>
<!ELEMENT alsoKnownAs (title,para1)>
<!ELEMENT motivation (title,(para1|umlDiagram)+)>
<!ELEMENT applicability (title,para+)>
<!ELEMENT structure (title,(para1|umlDiagram)+)>
<!ELEMENT participants (title,para1)>
<!ELEMENT collaborations (title,para1)>
<!ELEMENT consequences (title,para1)>
<!ELEMENT implementation (title,para1)>
<!ELEMENT sampleCode (title,para1)>
<!ELEMENT knownUses (title,para1)>
<!ELEMENT relatedPatterns (title,para1)>
```

```
<!ELEMENT heading (#PCDATA)>
<!ELEMENT hr EMPTY>
<!ELEMENT title (#PCDATA)>
<!ELEMENT para1 (#PCDATA|para2|link)+>
<!ELEMENT para2 (#PCDATA|link)+>
<!ELEMENT umlDiagram EMPTY>
<!ATTLIST umlDiagram image CDATA #REQUIRED>
<!ELEMENT link (#PCDATA)>
<!ATTLIST link href CDATA #REQUIRED>
```

Below is the second DTD for the PBR method.

```
<!-- pbr.dtd -->
<!ELEMENT pbr (heading, hr, systemName, developmentHistory,
classDoc, metricsDoc, patternDoc)>
<!ELEMENT systemName (title,para)>
<!ELEMENT developmentHistory (title,developer,year,organisation?)>
<!ELEMENT classDoc (title,link)>
<!ELEMENT metricsDoc (title,link)>
<!ELEMENT patternDoc (title,(patternInstance,c-class)*)>
<!ELEMENT title (para)>
<!ELEMENT developer (para)>
<!ELEMENT year (para)>
<!ELEMENT organisation (para)>
<!ELEMENT patternInstance (link,para)>
<!ELEMENT c-class (para)> <!ELEMENT para (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT hr EMPTY>
<!ELEMENT link (#PCDATA)>
<!ATTLIST link href CDATA #REQUIRED>
```

Based on the above two DTDs, for example, the following two XML documents can be produced.

```
<!-- abstractFactory.xml -->
<?xml version='1.0'?>
<!DOCTYPE pattern SYSTEM "pattern.dtd">
<?xml-stylesheet type="text/xsl" href="pattern.xsl"?>
<pattern>
<heading>Design Pattern Description</heading>
<hr/>
<name>
   <title>Name</title>
   <para1>Abstract Factory</para1>
</name>
<classification>
   <title>Classification</title>
   <para1>Object Creational</para1>
</classification>
<intent>
   <title>Intent</title>
   <para1>Provide an interface for creating families of
   related or dependent objects without specifying their
   concrete classes.
   </para1>
</intent>
<alsoKnownAs>
   <title>Also Known As</title>
   <para1>Kit </para1>
</alsoKnownAs>
<motivation>
   <title>Motivation</title>
   <para1>Consider a user interface toolkit that supports
   multiple look-and-feel standards, such as Motif and
   Presentation Manager. Different look-and-feels define
   different appearances and behaviors for user interface
   "widgets" like scroll bars, windows, and buttons. To be
   portable across look-and-feel standards, an application
   should not hard-code its widgets for a particular look
   and feel. Instantiating look-and-feel-specific classes
   of widgets throughout the application makes it hard to
   change the look and feel later.
   </para1>
   <para1> We can solve this problem by defining an
   abstract WidgetFactory class that declares an interface
   for creating each basic kind of widget. There's also an
   abstract class for each kind of widget, and concrete
   subclasses implement widgets for specific look-and-feel
   standards. WidgetFactory's interface has an operation
   that returns a new widget object for each abstract
   widget class. Clients call these operations to obtain
   widget instances, but clients aren't aware of the
   concrete classes they're using. Thus clients stay
   independent of the prevailing look and feel.
   </para1>
   <umlDiagram image='abfac109.gif'/>
   <para1>There is a concrete subclass of WidgetFactory for
```

```
    each look-and-feel standard. Each subclass implements the
    operations to create the appropriate widget for the look
    and feel. For example, the CreateScrollBar operation on
    the MotifWidgetFactory instantiates and returns a Motif
    scroll bar, while the corresponding operation on the
    PMWidgetFactory returns a scroll bar for Presentation
    Manager. Clients create widgets solely through the
    WidgetFactory interface and have no knowledge of the
    classes that implement widgets for a particular look and
    feel. In other words, clients only have to commit to an
    interface defined by an abstract class, not a particular
    concrete class.
    </para1>
    <para1>A WidgetFactory also enforces dependencies between
    the concrete widget classes. A Motif scroll bar should be
    used with a Motif button and a Motif text editor, and that
    constraint is enforced automatically as a consequence of
    using a MotifWidgetFactory.
    </para1>
</motivation>

<!-- Omission in the middle -->

<relatedPatterns>
    <title>Related Patterns</title>
    <para1>AbstractFactory classes are often implemented with
    <link href='factoryMethod.xml'>factory methods</link>,
    but they can also be implemented using
    <link href='prototype.xml'>prototype</link>.
    </para1>
    <para1>A concrete factory is often a
    <link href='singleton.xml'>singleton</link>.
    </para1>
</relatedPatterns>

</pattern>



<!-- pbr4Unidraw.xml -->

<?xml version='1.0'?>

<!DOCTYPE pbr SYSTEM "pbr.dtd">

<?xml-stylesheet type="text/xsl" href="pbr.xsl"?>

<pbr>

<heading>Pattern-Based Documentation</heading>

<hr/>

<systemName>
    <title>System Name</title>
    Unidraw
</systemName>

<developmentHistory>
    <title>Development History</title>
    <developer>John Vlissides</developer>
    <year>1993</year>
    <organisation>Stanford University, Palo Alto,
    California, USA</organisation>
```

```
</developmentHistory>
<classDoc>
   <title>Class Documentation</title>
   <link href='classDoc4Unidraw.txt'>Class Documentation
   of Unidraw</link>
</classDoc>
<metricsDoc>
   <title>Metrics Documentation</title>
   <link href='metricsDoc4Unidraw.html'>Software Metrics
   of Unidraw</link>
</metricsDoc>
<patternDoc>
   <title>Pattern Documentation</title>
   Detected pattern instances and their respective
   c-classes
   <patternInstance>
      <link href='chainOfResponsibility.xml'>Chain Of
      Responsibility</link>:
      <c-class>ComponentView</c-class>
   </patternInstance>
    <patternInstance>
      <link href='command.xml'>Command</link>:
      <c-class>Command</c-class>
   </patternInstance>
   <patternInstance>
      <link href='factoryMethod.xml'>Factory Method</link>:
      <c-class>Creator</c-class>
   </patternInstance>
   <patternInstance>
      <link href='iterator.xml'>Iterator</link>:
      <c-class>Iterator</c-class>
   </patternInstance>
   <patternInstance>
      <link href='mediator.xml'>Connector</link>:
      <c-class>Connector</c-class>
   </patternInstance>
   <patternInstance>
      <link href='memento.xml'>Memento</link>:
      <c-class>MoveCmd</c-class>
   </patternInstance>
   <patternInstance>
      <link href='observer.xml'>Observer</link>:
      <c-class>ComponentView</c-class>
   </patternInstance>
   <patternInstance>
      <link href='prototype.xml'>Prototype</link>:
      <c-class>GraphicCompTool</c-class>
   </patternInstance>
   <patternInstance>
      <link href='state.xml'>State</link>:
      <c-class>Tool</c-class>
   </patternInstance>
</patternDoc>
</pbr>
```

Two XSL documents were developed to display the two kinds of XML documents on InDelv. Below are them.

```
<!-- pattern.xsl -->
<?xml version='1.0'?>
<xsl:stylesheet
   xmlns:xsl='http://www.w3.org/XSL/Transform/1.0'
   xmlns:fo='http://www.w3.org/XSL/Format/1.0' result-ns='fo'>
   <xsl:template match='/'>
      <fo:display-sequence
         start-indent='4pt'
         end-indent='4pt'
         font-size='11pt'>
      <xsl:apply-templates/>
      </fo:display-sequence>
   </xsl:template>
   <xsl:template match='heading'>
      <fo:block
         font-size='18pt'
         font-weight='bold'
         space-before='18pt'
         space-after='12pt'>
         <xsl:apply-templates/>
      </fo:block>
   </xsl:template>
   <xsl:template match='hr'>
      <fo:display-rule
         start-indent='12pt'
         end-indent='12pt'
         rule-thickness='1.5pt'
         space-before='18pt'
         space-after='18pt'>
         <xsl:apply-templates/>
      </fo:display-rule>
   </xsl:template>
   <xsl:template match='title'>
      <fo:block
         font-size='15pt'
         font-weight='bold'
         space-before='18pt'
         space-after='12pt'>
         <xsl:apply-templates/>
      </fo:block>
   </xsl:template>
   <xsl:template match='para1'>
      <fo:block
         space-before='6pt'
         space-after='6pt'>
         <xsl:apply-templates/>
      </fo:block>
   </xsl:template>
   <xsl:template match='para2'>
      <fo:block
         space-before='6pt'
         space-after='6pt'
         start-indent='15pt'>
```

```
            <xsl:apply-templates/>
        </fo:block>
    </xsl:template>

    <xsl:template match='link'>
      <fo:simple-link
         external-destination='{@href}'
         color='rgb(0,0,255)'>
         <xsl:apply-templates/>
      </fo:simple-link>
    </xsl:template>
    <xsl:template match='umlDiagram'>
       <fo:display-graphic
          width='{@max-width}'
          height='{@max-height}'
          href='{@image}'>
          <xsl:apply-templates/>
       </fo:display-graphic>
    </xsl:template>
</xsl:stylesheet>


<!-- pbr.xsl -->

<?xml version='1.0'?>

<xsl:stylesheet
   xmlns:xsl='http://www.w3.org/XSL/Transform/1.0'
   xmlns:fo='http://www.w3.org/XSL/Format/1.0'
   result-ns='fo'>

   <xsl:template match='/'>
      <fo:display-sequence
         start-indent='4pt'
         end-indent='4pt'
         font-size='11pt'>
         <xsl:apply-templates/>
      </fo:display-sequence>
   </xsl:template>

   <xsl:template match='heading'>
      <fo:block
         font-size='18pt'
         font-weight='bold'
         space-before='18pt'
         space-after='12pt'>
         <xsl:apply-templates/>
      </fo:block>
   </xsl:template>

   <xsl:template match='hr'>
      <fo:display-rule
         start-indent='12pt'
         end-indent='12pt'
         rule-thickness='1.5pt'
         space-before='18pt'
         space-after='18pt'>
         <xsl:apply-templates/>
      </fo:display-rule>
   </xsl:template>

   <xsl:template match='title'>
```

```
      <fo:block
         font-size='15pt'
         font-weight='bold'
         space-before='18pt'
         space-after='12pt'>
         <xsl:apply-templates/>
      </fo:block>
   </xsl:template>

   <xsl:template match='patternInstance'>
      <fo:block
         space-before='6pt'
         space-after='6pt'>
         <xsl:apply-templates/>
      </fo:block>
   </xsl:template>

   <xsl:template match='link'>
      <fo:simple-link
         external-destination='{@href}'
         color='rgb(0,0,255)'>
         <xsl:apply-templates/>
      </fo:simple-link>
   </xsl:template>

</xsl:stylesheet>
```

Bibliography

[1] N. Akima and F. Ooi. Industrializing software development: A Japanese approach. *IEEE Software*, pages 13–22, March 1989.

[2] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, New York, 1979.

[3] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language*. Oxford University Press, New York, 1977. with Max Jacobson and Ingrid Fiksdahl-King and Shlomo Angel.

[4] Apple Computer, Inc. Macintosh programmers workshop Pascal 3.0 reference, 1989.

[5] Brad Appleton. Patterns and software: Essential concepts and terminology. < www.enteract.com/~bradapp/docs/patterns-intro.html> , An earlier revision of this paper appeared in the May 1997 Object Magazine Online (Vol. 3, No. 5)., 1998.

[6] Felix Bachman, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau. Volume II: Technical Concepts of Component-Based Software Engineering. Technical Report CMU/SEI-2000-TR-008, ESC-TR-2000-007, Software Engineering Institute, Carnegie Mellon University, May 2000.

[7] B. Barnes and T. Bollinger. Making software reuse cost effective. *IEEE Software*, pages 13–24, 1991.

[8] Victor R. Basili. GQM approach has evolved to include models. *IEEE Software*, 11(1):8, January 1994. Letter to the editor.

[9] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. Experience Factory. In John J. Marciniak, editor, *Encyclopedia of Software Engineering*, volume 1, pages 469–476. John Wiley & Sons, 1994.

[10] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. Goal Question Metric Paradigm. In John J. Marciniak, editor, *Encyclopedia of Software Engineering*, volume 1, pages 528–532. John Wiley & Sons, 1994.

[11] Victor R. Basili and Harlan Mills. Understanding and documenting programs. *IEEE Transactions on Software Engineering*, 8(3):270–283, May 1982.

[12] Victor R. Basili and Richard W. Selby, Jr. Data collection and analysis in software research and management. In *Proceedings of the American Statisti-*

cal Association and Biometric Society Joint Statistical Meeting, pages 21–30, Philadelphia, August 1984.

[13] Victor R. Basili and D. M. Weiss. A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering*, SE-10(6):728–738, November 1984.

[14] Don Batory. Intelligent components and software generators. Technical Report CS-TR-97-06, University of Texas, Austin, April 1, 1997.

[15] K. Beck and R. E. Johnson. Patterns generate architectures. In *ECOOP'94*, pages 139–149, 1994. LNCS 821.

[16] Steve Berczuk. Finding solutions through pattern languages. *IEEE Computer*, 27(12):75–76, December 1994.

[17] J. Bieman and S. Karunanithi. Candidate reuse metrics for object-oriented and ada software. In *Proceedings of IEEE-CS First International Software Metrics Symposium*, 1993.

[18] Ted Biggerstaff. *Software Reusability, Concepts and Models*, volume I, page xv. ACM Press, 1989.

[19] Peter J. Biggs. A survey of object-oriented methods. Technical Report 6/95, Department of Computer Science, University of Durham, 1995.

[20] Peter J. Biggs. *Automating Reuse Support in a Small Company.* PhD thesis, Durham University, August 1998.

[21] Dines Bjørner. On the use of formal methods in software development. In *Proc. of 9th International Conference on Software Engineering*, pages 17–29. IEEE, April 1987.

[22] B. W. Boehm. The high cost of software. In E. Horowitz, editor, *Practical Strategies for Developing Large Software Systems.* Addision-Wesley, Reading, MA, 1975.

[23] G. Booch. *Object-Oriented Analysis and Design.* Benjamin Cummings, 2nd edition, 1994.

[24] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, Reading, Massachusetts, USA, 1999.

[25] Jan Bosch. Design patterns frameworks: On the issue of language support. In *ECOOP Workshops*, pages 133–136, 1997.

[26] Jan Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming*, 11(2):18–32, 1998.

[27] Kyle Brown. Design reverse-engineering and automated design pattern detection in Smalltalk. Master's thesis, Department of Computer Engineering, North Carolina State University, 1996.

[28] Tim Bull. *Software Maintenance by Program Transformation in a Wide Spectrum Language*. PhD thesis, School of Engineering and Computer Science, University of Durham, 1994.

[29] Dan L. Burk. Copyrightable functions and patentable speech. *Communications of the ACM*, 44(2):69–75, February 2001.

[30] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A system of patterns - pattern oriented software architecture*. Wiley, 1996.

[31] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[32] Elliot J. Chikofsky and James H. Cross. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[33] Congress of the United States, Office of Technology Assessment. Intellectual property rights in an age of electronics and information. Technical report, Washington, D.C.: U.S. Government Printing Office, 1986.

[34] L. L. Constantine and E. Yourdon. *Structured Design*. Prentice Hall, Englewood Cliffs, NJ, 1979.

[35] James O. Coplien. A generative development - process pattern language. In J. Coplien and D. Schmidt, editors, *Pattern Languages of Program Design*, pages 183–237. Addison-Wesley, 1995.

[36] Yan-David Erlich. Implementing design patterns as languages constructs. *ACM SIGPLAN Notices*, 34(1):348–348, January 1999.

[37] Norman Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach.* International Thomson Computer Press, London, UK, second edition, 1997.

[38] N. T. Fletton and M. Munro. Redocumenting software systems using hypertext technology. In *Proceedings of the International Conference on Software Maintenance 1988*, pages 54–59. IEEE, IEEE Computer Society Press, 1988.

[39] Nigel T. Fletton. A hypertext approach to browsing and documenting software. In *HYPERTEXT II: State of the Art*, Prototypes, pages 193–204. Intellect, Inc., 1989.

[40] Marcus Fontoura, Wolfgang Pree, and Bernhard Rumpe. UML-F: A modeling language for object-oriented frameworks. In E. Bertino, editor, *ECOOP 2000—Object-Oriented Programming*, volume 1850 of *Lecture Notes in Computer Science*, pages 63–82. Springer, 2000.

[41] Brian Foote. A fractal model of the lifecyle of reusable objects. OOPSLA '91 Workshop on Reuse, Ottawa, Ontario, Canada, 1991.

[42] Gary Ford. Lecture notes on engineering measurement for software engineers. SEI educational materials package CMU/SEI-93-EM-9, Carnegie Mellon University, Software Engineering Institute, 1993.

[43] Martin Fowler. *Analysis patterns: reusable object models.* Addison Wesley Longman, Inc, 1997.

[44] William Frakes and Carol Terry. Software reuse: Metrics and models. *ACM Computing Surveys*, 28(2):415–435, June 1996.

[45] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, Massachusetts, 1994.

[46] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume 1, pages 1–40. World Scientific Publishing Company, 1993.

[47] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39, Singapore, 1993. World Scientific Publishing Company.

[48] Edward Stewart Garnett. *Software Reclamation: Upgrading Code for Reusability.* PhD thesis, Lancaster University, September 1990.

[49] K. Geary. Practical problems in introducing software reuse, May 1987. IEE Colloquium on Reusable Software Components.

[50] Joseph Gil and David H. Lorenz. Design patters vs. language design. In J. Bosch and S. Mitchell, editors, *Object-Oriented Technology: ECOOP'97 Workshop Reader*, number 1357 in Lecture Notes in Computer Science, pages 108–111. Language Support for Design Patterns and Frameworks Workshop Proceedings, Jyväskylä, Finland, Springer Verlag, June 9-13 1997.

[51] C. F. Goldfarb. *The SGML Handbook.* Oxford University Press, 1990.

[52] I. S. Graham and L. Quin. *XML Specification Guide.* John Wiley & Sons, Inc., 1999.

[53] Martin L. Griss. Implementing product-line features with component reuse. In William B. Frakes, editor, *Software Reuse: Advances in Software Reusability*, volume 1844 of *Lecture Notes in Computer Science*, pages 137–152. Springer, 2000. 6th International Conference on Software Reuse (ICSR-6).

[54] S. Hallsteinsen and M. Paci. *Experiences in Software Evolution and Reuse: Twelve Real World Projects.* Springer-Verlag, 1997.

[55] Sallie M. Henry and Dennis G. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, September 1981.

[56] Ivar Jacobson, Magnus Christerson, Patrk Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering – A Use Case Driven Approach.* Addison-Wesley, Wokingham, England, 1992.

[57] Ralph E. Johnson. Documenting Frameworks using Patterns. In *Proceedings of the OOPSLA '92 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 63–76, October 1992. Published as ACM SIGPLAN Notices, volume 27, number 10.

[58] T. C. Jones. Reusability in programming: A survey of the state of the art. *IEEE Transactions on Software Engineering*, 10(5):488–494, September 1984.

[59] J. van Katwijk and E. M. Dusink. Reusable software and software components. In R. J. Gautier and P. J. L. Wallis, editors, *Software Reuse with Ada*, pages 15–22. Peter Peregrinus Ltd., 1990.

[60] Rudolf K. Keller, Reinhard Schauer, Sebastien Robitaille, and Patrick Page. Pattern-based reverse-engineering of design components. In *Proceedings of the 21st International Conference on Software Engineering*, pages 226–235. ACM Press, May 1999.

[61] Norman L. Kerth and Ward Cunningham. Using patterns to improve our architectural vision. *IEEE Software*, pages 53–59, January 1997.

[62] H. Kim and C. Boldyreff. An approach to increasing software component reusability in Ada. *Lecture Notes in Computer Science*, 1088:89–100, 1996.

[63] Hyoseob Kim. Ada code reuse guidelines for design-for-reuse. Master's thesis, Department of Computer Science, University of Durham, 1996.

[64] Hyoseob Kim and C. Boldyreff. Software reusability issues in code and design. *ACM SIGADA Ada Letters*, 17(6):91–97, November/December 1997.

[65] Donald E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992.

[66] P. Kogut. Design reuse: Chemical engineering vs. software engineering. *Software Engineering Notes*, 20(5):73–77, 1995.

[67] Oh Cheon Kwon. *A process model of maintenance with reuse : an investigation and an implementation.* PhD thesis, Department of Computer Science, University of Durham, 1998.

[68] K. Lano and N. Malik. Reengineering legacy applications using design patterns. In *Proceedings of the 8th International Workshop on Software Technology and Engineering Practice*, London, UK, July 1997.

[69] Kevin Lano, Juan Bicarregui, and S. Goldsack. Formalising design patterns. In David Duke and Andy Evans, editors, *1st BCS-FACS Northern Formal Methods Workshop, Ilkley, UK*, Electronic Workshops in Computing. Springer-Verlag, 1996.

[70] Anthony Lauder and Stuart Kent. Precise visual specification of design patterns. In Eric Jul, editor, *ECOOP 1998—Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 114–134. Springer, 1998.

[71] David B. Leblang and Gordon D. McLean, Jr. Configuration management for large-scale software development efforts. In *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large*, pages 122–127, Harwichport, Massachusetts, 1985.

[72] Ted Lewis, Larry Rosenstein, Wolfgang Pree, Andre Weinand, Erich Gamma, Paul Calder, Glenn Andert, John Vlissides, and Kurt Schmucker. *Object-Oriented Application Frameworks*. Prentice-Hall, 1995.

[73] Mark A. Linton and Paul R. Calder. The design and implementation of InterViews. In Jim Waldo, editor, *The Evolution of C++: Language Design in the Marketplace of Ideas*, pages 75–86, Berkeley, CA, USA and Cambridge, MA, USA, 1993. USENIX and MIT Press. Editor: Jim Waldo.

[74] Mark A. Linton, Paul R. Calder, and John M. Vlissides. InterViews: A C++ graphical interface toolkit. Technical Report CSL-TR-88-358, Stanford University, Computer Systems Lab, July 1988.

[75] Stanley B. Lippman. *C++ Primer*. Addison-Wesley, Reading, Mass., 1989.

[76] Mark Lorenz. *Object-oriented software metrics: a practical guide*. PTR Prentice Hall, 1994.

[77] Simon Lucas. Optimisation of the similarity analyser. Master's thesis, Department of Computing, City University, London, U.K., November 1996.

[78] Robert Martin. Discovering patterns in existing applications. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Langauges of Program Design*, pages 365–393. Addison Wesley, 1995.

[79] Y. Matsumoto et al. SWB system: A software factory. *Software Engineering Environments*, pages 305–314, 1981.

[80] Sky Matthews and Carl Grove. Applying object-oriented concepts to documentation. In *ACM Tenth International Conference on Systems Documentation*, pages 265–271, 1992.

[81] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976.

[82] John A. McDermid. Introduction and overview to part II, methods, techniques and technology. In John A. McDermid, editor, *Software Engineer's Reference Book*. Butterworth-Heinemann, June 1990.

[83] J. McGregor and D. Sykes. *Object-Oriented Software Development: Engineering Software for Reuse*. Van Nostrand Reinhold, New York, 1992.

[84] M. D. McIlroy. Mass produced software components. In P. Naur, B. Randell, and J. N. Buxton, editors, *Proceedings of NATO Conference on Software Engineering*, pages 88–98, New York, 1969. Petrocelli/Charter.

[85] Eugene S. Meieran. 21st century semiconductor manufacturing capabilities. *Intel Technology Journal*, 1998. 4th Quarter '98.

[86] Bertrand Meyer. *Object-Oriented Software Construction.* Series in Computer Science. Prentice Hall, Englewood Cliffs, NJ, 1988.

[87] Hafedh Mili, Odile Marcotte, and Anas Kabbaj. Intelligent component retrieval for software reuse. In *Proceedings of the Third Maghrebian Conference on Artificial Intelligence and Software Engineering*, pages 101–114, Rabat, Morocco, April 1994.

[88] G. Moore. Cramming more components onto integrated circuits. *Electronics*, pages 114–117, April 1965.

[89] Hausi A. Müller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, December 1993.

[90] P. Naur and B. Randell, editors. *Software Engineering: A Report on a Conference sponsored by the NATO Science Committee.* NATO, 1969.

[91] J. Neighbors. The Draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, 10(5):564–573, September 1984.

[92] Object Management Group, Inc. OMG Unified Modeling Language Specification. Version 1.3, June 1999.

[93] William F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[94] Robert Orfali, Dan Harkey, and Jeri Edwards. *Instant CORBA*. John Wiley, 1997.

[95] C. Papidimitriou and K. Steiglitz. *Combinatorial Optimisation: Algorithms and Complexity*. Prentice Hall, Englewood Cliffs, NJ, 1982.

[96] Mark C. Paulk, Bill Curtis, Mary Beth Chrissis, and Charles V. Weber. Capability maturity model for software, version 1.1. Technical Report CMU/SEI-93-TR-024, ESC-TR-93-177, Software Engineering Institute, Carnegie Mellon University, February 1993.

[97] Mark C. Paulk, Charles V. Weber, Suzanne M. Garcia, Mary Beth Chrissis, and Marilyn Bush. Key practices of the capability maturity model, version 1.1. Technical Report CMU/SEI-93-TR-025, ESC-TR-93-178, Software Engineering Institute, Carnegie Mellon University, February 1993.

[98] A. Perlis, F. Sayward, and M. Shaw. The role of metrics in software and software development. In *Software Metrics: An Analysis and Evaluation*, pages 1–4, Cambridge, Massachusetts, 1981. The MIT Press.

[99] D. Perry. System compositions and shared dependencies. Software Configuration Management: ICSE'96 SCM-6 Workshop Selected Papers, Berlin, April 25–26,1996, Published as Lecture Notes in Computer Science, 1167:139–153. Springer.

[100] Cuno Pfister and Clemens Szyperski. Why objects are not enough. In *Proceedings, International Component Users Conference*, Munich, Germany, 1996. SIGS.

[101] J. Poulin. *Measuring Software Reuse*. Addison Wesley, 1996.

[102] Lutz Prechelt. An experiment on the usefulness of design patterns: detailed description and evaluation. Technical Report iratr-1997-9, Universität Karlsruhe, Institut für Programmstrukturen und Datenorganisation, 1997.

[103] Lutz Prechelt and Barbara Unger. A series of controlled experiments on design patterns: Methodology and results. In *Proc. Softwaretechnik'98, GI Conference, Paderborn*, pages 53–60, 1998.

[104] Lutz Prechelt, Barbara Unger, and Michael Philippsen. Documenting design patterns in code eases program maintenance. In *Proceedings of ICSE Workshop on Process Modeling and Empirical Studies of Software Evolution*, Boston, MA, May 1997.

[105] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, third edition, 1992.

[106] R. Prieto-Diaz. Status report: Software reusability. *IEEE Software*, pages 61–66, May 1993.

[107] M. Ramachandran. *An Investigation into Tool Support for the Development of Reusable Software*. PhD thesis, Lancaster University, 1992.

[108] D. J. Robson, K. H. Bennett, B. J. Cornelius, and M. Munro. Approaches to program comprehension. *The Journal of Systems and Software*, 14(2):79–84, February 1991.

[109] Kenneth Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill Publishing Company, June 1999 (4th ed).

[110] S. Rugaber. Program comprehension. In *Encyclopedia of Computer Science and Technology*, volume 35(20), pages 341–368. Marcel Dekker, Inc., New York, 1995.

[111] James Rumbaugh, Michael Blaha, William Premerlani, frederick Eddy, and William Lorenson. *Object-Oriented Modelling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[112] Pamela Samuelson. Intellectural property for information age. *Communications of the ACM*, 44(2):67–68, February 2001.

[113] Pamela Samuelson and Kevein Deasy. Intellectual property protection for software. SEI Curriculum Module SEI-CM-14-2.1, Carnegie Mellon University, Software Engineering Institute, July 1989.

[114] Reinhard Schauer and Rudolf K. Keller. Pattern visualization for software comprehension. In *Proceedings of the 6th International Workshop on Progrma Comprehension*, pages 4–12. IEEE Computer Society, June 1998.

[115] Douglas C. Schmidt. Using design patterns to guide the development of reusable object-orient software. Position statement for the ACM Workshop on Strategic Directions in Computing Research, MIT, June 1996.

[116] Roger Sessions. *COM and DCOM: Microsoft's Vision for Distributed Objects*. John Wiley, 1998.

[117] Mary Shaw. Prospects for an engineering discipline of software. Technical Report CMU/SEI-90-TR-20, ESD-TR-90-221, Software Engineering Institute, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213-3809, September 1990.

[118] Mary Shaw and David Garlan. *Software Architecture: Perspective on an Emergining Discipline*. Prentice Hall, 1996.

[119] Forrest Shull, Walcélio L. Melo, and Victor R. Basili. An inductive method for discovering design patterns from object-oriented software systems. Technical Report CS-TR-3597, UMIACS-TR-96-10, Computer Science Department/Institute for Advanced Computer Studies, University of Maryland, 1996.

[120] Paul Singleton and Pearl Brereton. A case for declarative programming-in-the-large. Technical Report TR93-14, Department of Computer Science, Keele University, Keele, UK, June 1993.

[121] Dennis B. Smith. Y2K: Organizational issues, November 1997. Presented at the 1997 IBM CAS Conferance (CASCON '97), Toronto, Canada.

[122] I. Sommerville. Software reuse: Potential and problems. Technical Report CS-SE-1-87, Department of Computing, University of Lancaster, 1987.

[123] I. Sommerville. *Software Engineering*, chapter 30, pages 571–588. Addison-Wesley, fourth edition, 1992.

[124] Ian Sommerville. Software reuse courses. Software Reuse Course Slides, 1994.

[125] G. Spanoudakis and K. Kassis. An evidential framework for diagnosing the significance of inconsistencies in UML models. In *Proceedings of the International Conference on Software: Theory and Practice*, pages 152–162, Bejing, China, August 2000.

[126] George Spanoudakis and Panas Constantopoulos. Similarity for analogical software reuse: A computational model. In A. G. Cohn, editor, *Proceedings of the Eleventh European Conference on Artificial Intelligence*, pages 18–22, Chichester, August 8–12 1994. John Wiley and Sons.

[127] George Spanoudakis and Anthony Finkelstein. Reconciling requirements: a method for managing interference, inconsistency and conflict. *Annals of Software Engineering*, 3:433–457, 1997. Software Requirements Engineering.

[128] George Spanoudakis, Anthony Finkelstein, and David Till. Overlaps in requirements engineering. *Automated Software Engineering: An International Journal*, 6(2):171–198, April 1999.

[129] Perdita Stevens and Rob Pooley. Systems reengineering patterns. In *Proceedings of the ACM SIGSOFT 6th International Symposium on the Foundations of Software Engineering (FSE-98)*, volume 23, 6 of *Software Engineering Notes*, pages 17–23, New York, November 3–5 1998. ACM Press.

[130] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, Reading, MA, 1994.

[131] Gerson Sunyé, Alain Le Guennec, and Jean-Marc Jézéquel. Design patterns application in UML. In E. Bertino, editor, *ECOOP 2000—Object-Oriented Programming*, volume 1850 of *Lecture Notes in Computer Science*, pages 44–62. Springer, 2000.

[132] E. B. Swanson. The dimensions of maintenance. In *Proceedings of the Second International Conference on Software Engineering*, pages 492–497. IEEE, October 1976.

[133] Clemens Szyperski. Component-oriented programming: A refined variation on object-oriented programming. *The Oberon Tribune*, 1(2), 1995.

[134] Talignet Inc. Leveraging object-oriented frameworks. A Taligent White Paper, 1996.

[135] S. R. Tilley. Documenting-in-the-large vs. documenting-in-the-small. In *Proceedings of CASCON 1993*, pages 1083–1090, October 1993.

[136] Scott R. Tilley and Dennis B. Smith. Perspectives on legacy system reengineering. Draft Version 0.3, Reengineering Center, Software Engineering Institute, Carnegie Mellon University, 1995.

[137] W. Tracz, L. Coglianese, and P. Young. A domain-specific software architecture engineering process outline. *ACM SIGSOFT Software Engineering Notes*, 18(2):40–49, 1993.

[138] John M. Vlissides. *Generalised Graphical Object Editing*. PhD thesis, Stanford University, June 1990.

[139] John M. Vlissides and Mark A. Linton. Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems*, 8(3):237–268, July 1990.

[140] Jeffrey M. Voas. Certifying off-the-shelf software components. *Computer*, 31(6):53–59, 1998.

[141] Anneliese von Mayrhauser and A. Marie Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, August 1995.

[142] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modelling with UML*. Object Technology Series. Addison-Wesley, Reading, Mass., 1999.

[143] Arthur H. Watson and Thomas J. McCabe. Structured testing: A testing methodology using the cyclomatic complexity metric. NIST Special Publication 500-

235, Computer Systems Laboratory, National Institute of Standards and Technology, Gaithersburg, MD, September 1996.

[144] A. Weinand, E. Gamma, and R. Marty. Design and implementation of ET++, a seamless object-oriented application framework. *Structured Programming*, 10(2):63–87, 1989.

[145] Kenny Wong, Scott R. Tilley, Hausi A. Muller, and Margaret-Anne D. Storey. Structural redocumentation: a case study. *IEEE Software*, 12(1):46–54, January 1995.

[146] A. Zisman. An overview of XML. *Computing & Control Engineering Journal*, 11(4), 2000.

Glossary

- **abstraction** The essential characteristics of an entity that distinguish it from all other kinds of entities. An abstraction defines a boundary relative to the perspective of the viewer.

- **aggregation** A special form of association that specifies a whole-part relationship between the aggregate (the whole) and a component (the part).

- **architecture** The set of significant decisions about the organisation of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behaviour as specified in the collaborations among those elements, the composition of these structural and behavioural elements into progressively larger subsystems, and the architectural style that guides this organisation.

- **artifact** A piece of information that is used or produced by a software development process.

- **association** A structural relationship that describes a set of links, in which a link is a connection among objects; the semantic relationship between two or more classifiers that involves the connections among their instances.

- **black-box reuse** A style of reuse based on object composition. Composed objects reveal no internal details to each other and are thus analogous to "black boxes".

- **class** A class defines an object's interface and implementation. It specifies the object's internal representation and defines the operations the object can perform.

- **class diagram** A diagram that depicts classes, their internal structure and operations, and the static relationships between them.

- **collaboration** A society of roles and other elements that work together to provide some cooperative behaviour that is bigger than the sum of all its parts; the specification of how an element, such as a use case or an operation, is realised by a set of classifiers and associations playing specific roles and used in a specific way.

- **component** A physical and replaceable part of a system that conforms to and provides the realisation of a set of interfaces.

- **context** A set of related elements for a particular purpose, such as to specify an operation.

- **coupling** The degree to which software components depend on each other.

- **Design Pattern Recovery (DPR)** The process of a program in an effort to create a representation of the program at a higher level of abstraction than source code. The recovered abstraction is in the format of design pattern.

- **Documentation-in-the-Large (DitL)** Documentation in the large scale where the overall structure and behaviour of a system are considered more important than localised descriptions of modules. DitL and DitS are not contradictory but complementary.

- **Documentation-in-the-Small (DitS)** Documentation in the small scale. As the size and complexity of systems increase, DitL is being considered more critical to the success of software development and maintenance projects than DitS.

- **Document Type Definition (DTD)** The grammar by which an XML document is defined. In other words, DTD specifies the structure of an XML document and how its content is nested.

- **domain** An area of knowledge or activity characterised by a set of concepts and terminology understood by practitioners in that area.

- **encapsulation** The result of hiding a representation and implementation in an object. The representation is not visible and cannot be accessed directly from outside the object. Operations are the only way to access and modify an object's representation.

- **eXtensible Markup Language (XML)** An initiative from the W3C defining an "extremely simple" dialect of SGML suitable for use on the World-Wide Web.

- **eXtensible Style Language (XSL)** A language used to create stylesheets for XML, similar to CSS (Cascading Style Sheets) that are used for HTML. In XML, content and presentation are separate. XML tags do not indicate how they should be displayed. An XML document has to be formatted before it can be read, and the formatting is usually accomplished with stylesheets. Stylesheets consist of formatting rules for how particular XML tags affect the display of a document on a computer screen or a printed page.

- **forward engineering** The process of transforming a model into code through a mapping to a specific implementation language.

- **framework** An architectural pattern that provides an extensible template for applications within a domain.

- **generalisation** Another name for inheritance.

- **inheritance** A relationship that defines one entity in terms of another. Class inheritance defines a new class in terms of one or more parent classes. The new class inherits its interface and implementation from its parents. The new class is called a subclass or (in C++) a derived class. Class inheritance combines interface inheritance and implementation inheritance. Interface inheritance defines a new interface in terms of one or more existing interfaces. Implementation

inheritance defines a new implementation in terms of one or more existing implementations.

- **instance** A concrete manifestation of an abstraction; an entity to which a set of operations can be applied and that has a state that stores the effects of the operations.
- **interaction diagram** A diagram that shows the flow of requests between objects.
- **level of abstraction** One place in a hierarchy of abstractions ranging from high levels of abstraction (very abstract) to low levels of abstraction (very concrete).
- **message** A specification of a communication between objects that conveys information with the expectation that activity will ensue; the receipt of a message instance is normally considered an instance of an event.
- **metaclass** A class whose instances are classes.
- **model** A simplification of reality, created in order to understand the system being created better; a semantically closed abstraction of a system.
- **object** A run-time entity that packages both data and the procedures that operate on that data.
- **object composition** Assembling or composing objects to get more complex behavior.

- **Object Constraint Language (OCL)** A formal language used to express side effect-free constraints.
- **object diagram** A diagram that depicts a particular object structure at run-time.
- **pattern** A common solution to a common problem in a given context.
- **Pattern-Based Redocumentation (PBR)** Redocumentation of a system on the basis of recovered or detected patterns.
- **polymorphism** The ability to substitute objects of matching interface for one another at run-time.
- **product** The artifacts of development, such as models, code, documentation, and work plans.
- **Programming-in-the-Large (PitL)** Programming in the large scale where sound engineering principles like reuse, measurement and CASE tools need to be applied for the success of the projects.
- **Programming-in-the-Small (PitS)** Programming in the small scale where systems are built by a person or, at most, a small group of people.
- **relationship** A semantic connection among elements.
- **requirement** A desired feature, property, or behaviour of a system.
- **reverse engineering** The process of transforming code into a model through a mapping from a specific implementation language.

- **subsystem** A grouping of elements of which some constitute a specification of the behaviour offered by the other contained elements.

- **system** Possibly decomposed into a collection of subsystems, a set of elements organised to accomplish a specific purpose and described by a set of models, possibly from different viewpoints.

- **Unified Modelling Language (UML)** A language for visualising, specifying, constructing, and documenting the artifacts of a software-intensive system.

- **white-box reuse** A style of reuse based on class inheritance. A subclass reuses the interface and implementation of its parent class, but it may have access to otherwise private aspects of its parent.

- **XML Linking Language (XLink)** An XML application that expands the way hyperlinks can be used. XLink makes it possible to target a specific section of a document, and adds other options to make linking easier.

- **XML Path Language (XPath)** A language that describes a way to locate and process items in XML documents by using an addressing syntax based on a path through the document's logical structure or hierarchy.

- **XML Pointer Language (XPointer)** is a language for locating data within an XML document based on properties such as location within the document, character content, and attribute values.